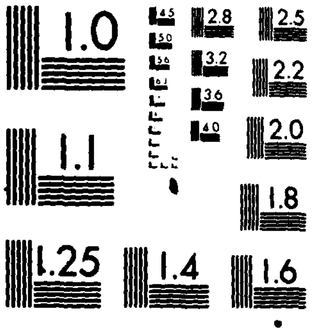
AD-A112 542   CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER --ETC F/G 9/2
ON A HIGH-PERFORMANCE VLSI SOLUTION TO DATABASE PROBLEMS.(U)
AUG 81 S W SONG   N00014-76-C-0370
UNCLASSIFIED CMU-CS-81-142   NL

1 OF 2
AD A112542

11254



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963-A

CMU-CS-81-142

12

On a High-Performance VLSI Solution to Database Problems

Siang Wun Song

August 1981

DEPARTMENT
of
COMPUTER SCIENCE

Carnegie-Mellon University

CMU-CS-81-142 12

On a High-Performance VLSI Solution to Database Problems

Siang Wun Song

August 1981

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in
Computer Science at Carnegie-Mellon University

This research was supported in part by the Office of Naval Research under Contracts N00014-76-C-0370, NR 044-422, and N00014-80-C-0236, NR 048-659, in part by the National Science Foundation under Grant MCS 78-236-76, and in part by the Defense Advanced Research Projects Agency under Contract F33615-78-C-1551 (monitored by the Air Force Office of Scientific Research). The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the United States Government. The author was supported in part by FAPESP (1976-1980), Fundação de Amparo à Pesquisa do Estado de São Paulo, Brazil, under Contract No. 76-517, in part by CNPq (1980-1981), Conselho Nacional de Desenvolvimento Científico e Tecnológico, Brazil, under Contract 200.402-79-CC, in part by the Institute of Mathematics and Statistics of the University of São Paulo, Brazil, and in part by the Department of Computer Science, Carnegie-Mellon University.

Table of Contents

List of Figures

List of Tables

to my wife Ping Ping

Acknowledgments

I wish to thank my adviser H. T. Kung for his constant encouragement and advice throughout my entire study and research work. I would also like to thank Jon Bentley, Chuck Eastman, and Bob Sproull for serving on the thesis committee and for their valuable feedback and dedicated reading of the manuscript.

Special thanks are due to Mike Foster, Phil Lehman, Kemal Oflazer, Bob Whiteside, and Z. M. You for very helpful discussions and suggestions. Also, I thank the entire CSD community for building and maintaining an enjoyable environment for doing research.

I am grateful to the University of São Paulo, FAPESP (Fundação de Amparo à Pesquisa do Estado de São Paulo), and CNPq (Conselho Nacional de Desenvolvimento Científico e Tecnológico) for their financial support.

Abstract

This thesis explores the design and use of custom-made VLSI hardware in the area of database problems. Our effort differs from most previous ones in that we search for structures and algorithms, directly implementable on silicon, for the solution of computation-intensive database problems. The types of target database systems include the general database management systems and the design database systems. The thesis deals mainly with database systems of the relational model. One common view concerning special-purpose hardware usage is that it performs a specific task. The proposed device is not a hardware solution to a specific problem, but provides a number of useful data structures and basic operations. It can be used to improve the performance of any sequential algorithm which makes extensive use of such data structures and basic operations. The design is based on a few basic cells, interconnected together in the form of a complete binary tree. The proposed device can handle all the basic relational operations: select, join, project, union, and intersection. With a special-purpose device of limited size attached to a host, the overall performance may ultimately be dictated by the I/O between the two sites. The ideal special-purpose device design is one that achieves a balance between computation and I/O. We propose a model to study the I/O complexity for sorting n numbers with any special-purpose hardware device of size s, and show a lower bound result of $\Omega(n \log n/\log s)$. We present an optimal design achieving this bound. An important finding is that for practical ranges on the quantity of data to be sorted, systolic sorting devices of small sizes can beat fast sequential sorting algorithms. To evaluate the theme that a data structure supporting a few basic operations can be useful, we examine a number of database problems all of which depend heavily on sort, join, and search operations. One problem is the detection of three-dimensional patterns of m points in a large structure of n points, with application in chemical databases. We present a sequential algorithm with worst-case time complexity of $\min\{O(n^2), O(n\ m^3)\}$. With a uniform distribution of points in the pattern, the time complexity is at most $O(m\ n)$. For the three-dimensional shape matching problem, where scale is also allowed, we give an $O(n^2)$ solution. We also propose a method of packaging certain large tree structures on chips. The packaging method requires only one type of fully utilized chip, which is not pin-bound. The wire lengths connecting component chips are shorter than those using the previously known layouts.

Chapter 1
Introduction

With recent technological advances in the VLSI circuitry, the chip capacity, or component count on a chip, is increasing at an astonishing rate [66, 74]. Both the opportunities and challenges regarding effective use of VLSI are tremendous. Considerable interest has been aroused in employing special-purpose hardware systems in application areas that demand good response time and throughput. Examples of application areas already being considered include signal and image processing, and graphics. This thesis explores the design and use of custom-made VLSI hardware in the area of database problems. Search for hardware-oriented solutions to database problems, in the context of design and implementation of the so-called database machines, has been around for more than ten years. Our focus here, however, differs from most previous efforts in that we search for structures and algorithms, directly implementable on silicon, for the solution of computation-intensive database problems. The proposed design is based on a few basic cells, interconnected in a simple and regular way. By replicating the basic cells, whose design constitutes the main design effort, a powerful special-purpose device can be obtained. This means that design costs, which typically constitute the dominant cost in special-purpose VLSI hardware systems, can be greatly reduced. The size of the custom-made device can be varied accordingly to match various performance goals. The proposed device can be used in two ways. A device of "large" size (to be made precise later) can handle all basic relational operations with substantial performance improvement over conventional methods. It may constitute a viable solution when chip capacity continues to increase. Another way is to use a device of "small" size as a sorting engine. We shall show that such a device can provide significant speed-ups in relation to fast sequential sorting algorithms. This solution is attractive because it implements an operation used in many database problems, and it can be implemented with current technology at a low cost. In either case the hardware is tailored to a specific application area and can thus be simpler in design than a general-purpose off-the-shelf computing device. The consequence is that we are likely to obtain a final product which is both compact in silicon area and able to provide good performance. Such a solution can thus be cost effective.

1.1 Goal and Motivation

The goal of the present research work is to investigate the feasibility of employing special-purpose VLSI hardware to solve compute-bound database problems. The types of target database systems include the general database management systems, as well as the so-called design database systems. The latter type of database system not only stores a model of some complex reality and provides primitive accesses to it, but also supports its design. Examples of design database systems include those used for architectural and mechanical designs. The thesis deals mainly with database systems of the relational model[1].

By organizing information in database systems, integration is achieved. An integrated database reduces the redundancy in the stored data. One important advantage is the sharing of data and software for their manipulation and retrieval among different applications and new ones to be developed. Because of these advantages, use of database systems has become widespread. More recent database systems offer such attractive features as automatic verification and maintenance of semantic integrity, usage of views as abstraction and authorization mechanisms, etc. The price for such features is the performance penalty a user must often pay. What is allowed in a practical database system is often limited by performance concerns. Operations such as the relational join and sort are computationally expensive. An example of a costly query is the conjunctive query [16], shown to include a great number of queries actually asked in practice. For example, the language Query-by-Example [96] is based on a core of conjunctive queries. An optimal implementation of conjunctive queries utilizes a great number of expensive relational joins. In design databases

[1] Appendix A contains a succinct introduction to the relational data model, as well as a brief discussion of the storage structures used in implementations of relational database systems.

such as those used for architectural and mechanical designs, costly spatial operations are often required. In masonry designs using object grammars [70], for example, the application of each production rule requires the complete solution of a complex three-dimensional shape matching problem. Details of this problem, as well as a number of other examples, will be examined in Chapter 6.

Semantic integrity verification and maintenance in design databases is discussed in [57, 58]. Integrity constraints in design databases may be very complex. Lafue [58] proposes delaying the maintenance of integrity until strictly necessary. As a consequence; violations of integrity are temporarily tolerated as long as they are known. Such measures reflect the performance concerns in design databases. Among the commercial database management systems, some kind of semantic integrity checking and maintenance has been implemented in System R [1, 2] and INGRESS [90]. Though some simple integrity constraints are negligible in cost, the user is warned against enforcing more complex controls unless he is willing to pay a considerable cost. GLIDE [27] is an example of a design database with data and control abstraction features of a high-level programming language. Some degree of semantic integrity verification is supported.

Another costly operation is the materialization of views. A view is a relation derived from existing relations, which may include base relations or derived relations. Nesting of views is therefore possible. Views constitute a powerful abstraction mechanism, as well as an authorization mechanism [29]. The creator of a view has the privileges to access, update (under certain conditions), and destroy it. It is also possible to *grant* some of these privileges to other users, and subsequently *revoke* the same privileges if desired. Views can also be used for another purpose. In order to enforce data independence and avoid redundancy, several kinds of normal forms have been proposed [20, 28]. In such normal forms, relations are broken into smaller parts which are then joined back together by the view mechanism. Materializing views may be costly when complex relational operations are involved [90]. In a recent report where System R is evaluated, Astrahan et al. [2] mention a performance penalty in executing very complex statements involving joins of several relations. They also point out that this performance degradation must be traded off against the advantages of normalization.

These considerations serve as motivation for the development and usage of special-purpose hardware to improve performance.

1.2 Methodology and Evaluation

A taxonomy and appraisal of database machine designs will be presented in Chapter 2. Such a study is related to the thesis in two ways. The appraisal will enable us to spot the strong and weak points of existing designs. For example, we will see that, with rare exceptions, previous works do not address the important issue of problem decomposition, that is, the partitioning of a large problem to be handled by a smaller piece of special-purpose hardware. The proposed taxonomy will provide us with the nomenclature needed to describe the context under which the proposed design stands. The taxonomy is based on several dimensions. One dimension is whether special-purpose logic is applied to secondary memory or primary memory, giving what we call *logic-enhanced secondary storage* or *primary storage* designs, respectively. Most previous designs belong to the first category, which is especially suitable for I/O-bound tasks. We shall focus mainly on designs that fall into the latter category, aimed at compute-bound tasks.

We shall consider a different view of special-purpose hardware usage. One common view is that a special-purpose device performs a well-defined and specific task such as filtering or two-dimensional convolution. The systolic architecture [56] was proposed as a solution to meet high-performance requirements for a compute-bound computational task that is *regular*, i. e., where repetitive computations are performed on a large set of data. Such is the case for example with respect to many signal and image processing computations. Nevertheless we would like to exploit usage of some kind of special-purpose hardware to handle computations that lack such regularity. The device we propose is not a hardware solution to a specific problem, but rather provides a number of useful data structures and basic operations. More specifically, it can be viewed as a hardware version of a heap, *priority queue*, or a data structure for searching in general. A sequential algorithm using complex data structures may spend a great amount of time in maintaining them. Overheads in book-keeping and control may be very expensive. This explains why certain asymptotically fast algorithms are not usable in practice, when the constant hidden in the O-notation makes the algorithm inappropriate for the actual problem size. The same overheads in the hardware counterpart are usually small. The performance of any sequential algorithm which makes extensive use of such data structures and the supported basic operations will thus improve with the aid of such a device. The price of this flexibility is that the device is not self-contained in that it needs the supervision of some controller which may be part of the interface system or the host.

One focus of paramount importance is the study of decomposition procedures by which a large problem is partitioned into subproblems to be handled by a smaller special-purpose device. As a result, it may be necessary to store and retrieve intermediate results before a computation is complete. The overall performance may ultimately be dictated by the I/O between the device and the host. We emphasize the importance of study of I/O complexity to achieve the optimal design where computation and I/O are balanced.

As a means of evaluation, we will show that such a device is indeed useful in database problems and cost effective. We anticipate here a summary of conclusions. We show that the proposed device can handle all the basic relational operations: select, join, project, union, and intersection. Furthermore, we show that a variety of problems are reducible to this set of basic operations. These include two costly three-dimensional pattern or shape detection problems and some others involving spatial operations. These problems arise in design databases and some special database systems. One result that may be of practical significance is a new method for packaging certain tree structures among chips. With regard to sorting, an important finding is that for practical ranges on the quantity of data to be sorted, systolic sorting devices of small sizes can beat fast sequential sorting algorithms.

1.3 Organization of the Thesis

Chapter 2 contains a taxonomy and appraisal of existing database machine designs. Chapter 3 describes a tree-structured device to perform sorting and some basic relational operations as project, join, union, etc. Partitioning mechanisms for handling large problems whose size exceeds that of the primary memory are discussed. Chapter 4 contains some implementation considerations. We present results on the packaging of large tree structures among chips, based on a layout result for a new structure called a *linearized tree*. We also discuss the design of the basic cells of the device, and propose a simple instruction set. Chapter 5 focuses on the importance of I/O on the overall system performance where a special-purpose device of limited size is used. Lower bound results on the I/O complexity for sorting are derived and an optimal design achieving this bound is shown. Chapter 6 examines five different problems that arise in special database systems, including detection of patterns of points and lines in three-space. The solutions are found to depend heavily on sort, join and search. Finally, concluding remarks, a summary of main results and contributions, and further works are presented in Chapter 7.

Chapter 2

A Taxonomy and Appraisal of Database Machine Designs

The purpose of this chapter is twofold. We survey existing database machine designs to identify their strengths and weaknesses, and propose a taxonomy which will then be used as a context for our research. This chapter is organized as follows. First we characterize the problem by identifying two bottlenecks. Then we describe the several dimensions the taxonomy is based on. At the risk of oversimplification, the proposed taxonomy attempts to group many seemingly differen: designs into a few categories by concentrating on their similarities. The remainder of the chapter can then be viewed as a detailed presentation of these categories, with a brief survey of previous database machine designs that fall into each category. Part of the appraisal of database machines is based on a paper by Langdon [60].

2.1 Problem Characterization

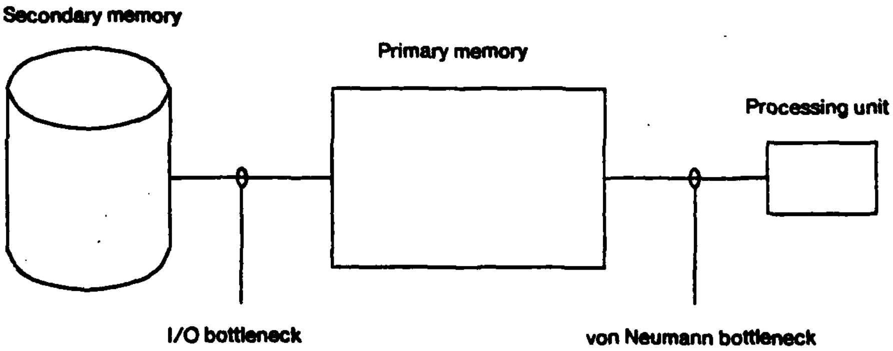


Figure 2-1: Two kinds of bottlenecks.

As shown in Figure 2-1, we can identify two potential bottlenecks, namely the I/O bottleneck and the so-called von Neumann bottleneck. Memory hierarchies exist because of economic reasons. If primary memory were inexpensive, there would be no need for secondary storage devices. Database files typically reside in the less expensive secondary memory and only needed portions are brought into primary memory for processing. Among the secondary storage devices, the magnetic disks have come into wide use. In moving-head disks, a comb-like assembly contains the read/write heads. All tracks under the read/write heads form a cylinder. To access data in a sector, the assembly first positions the read/write heads over the cylinder containing the desired track and sector. The time to do this is called the *seek time*. Having located the desired track, there is rotation delay for the needed sector to come under the read/write head. This time is the *latency time*. Finally there is the *transmission time* required to read or write a series of sectors. In fixed-head disks, every track has its own read/write head. No mechanical movement of the head assembly is involved and the seek time is essentially zero. Careful design of access paths and maintenance of appropriate indices help in reducing the number of disk accesses. To eliminate the need for access paths and indices, database machines with fast retrieval times have been proposed. Most database machine designs use the *logic-per-track* approach [84], allocating a read head, some search logic, and write head to each data track of rotating storage devices. Data read by the read head can be compared against some stored constant and modified, and subsequently written back to the track during the same revolution. Such designs have content addressability as their basic characteristic and can provide fast on-the-fly retrieval.

In a compute-bound task a data element participates in many operations. The best place to carry out such a task is inside the primary memory, because of its faster access speed. It was observed in [56] that in a conventional von Neumann machine, each operation typically fetches one or more operands from memory. Hence the amount of I/O (between the memory and the central processing unit) is proportional to the number of operations to be performed rather than the number of inputs required for the computation. The von Neumann bottleneck is in fact an I/O bottleneck in lesser scale. To reduce traffic through this bottleneck, many works have been done in the context of optimal register allocation or usage of cache memory. Studies of such solutions abound in the literature and fall outside the scope of the present work.

2.2 Dimensions of the Space of Database Machine Designs

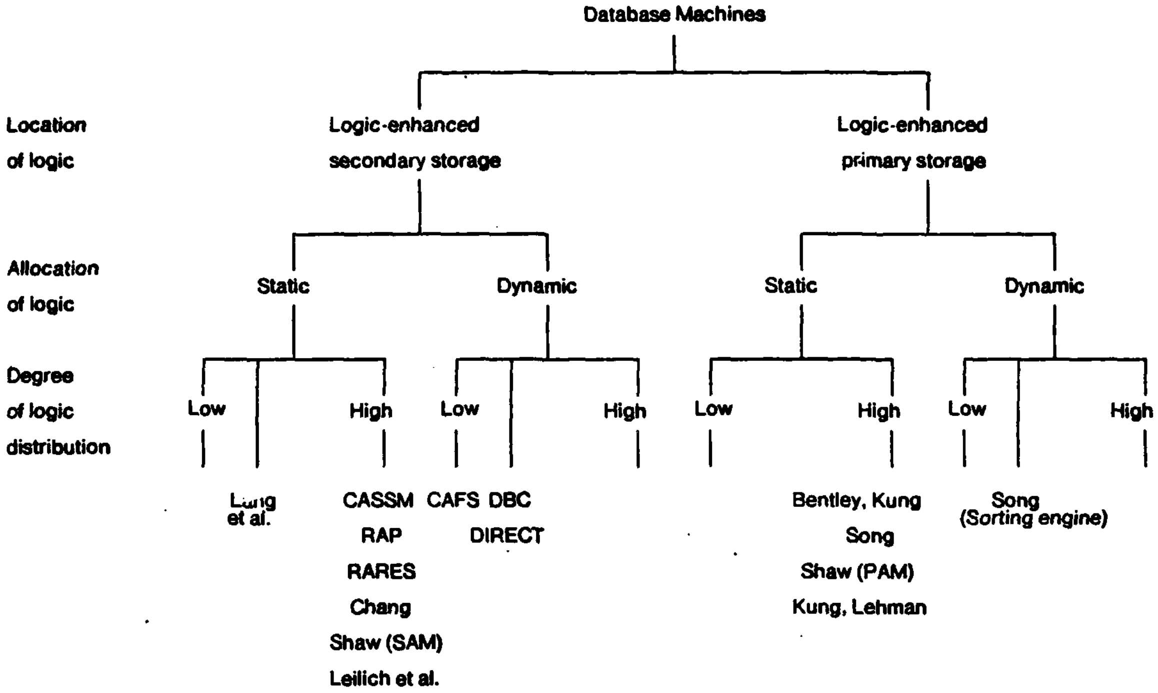


Figure 2-2: An overall framework.

The taxonomy is based on several dimensions. Depending on where special-purpose logic is applied, we have *logic-enhanced secondary* or *primary storage* designs. Most previous designs are variations of the first type. Another dimension along which designs can be classified is the way logic is allocated to storage units, whether *statically* or *dynamically*. A third dimension is the degree of distribution of logic among memory elements, defined in [83] as the number of storage elements associated with each processing unit. Along this dimension, we may have a wide spectrum of designs. High degree of logic distribution signifies faster computation rate, and this should be such that a balance between computation and data access rate is achieved. Classification of designs along this dimension is also important since it is related to the cost of a physical implementation. As an example, we can view a conventional von Neumann machine as a logic-enhanced primary storage device occupying the lowest end of the logic distribution spectrum. The allocation of logic is dynamic since one processing unit serves the entire memory. Figure 2-2 shows an overall framework, illustrated with a few particular designs. The reader can refer to this figure when reading the rest of this chapter. Its purpose is to illustrate the several categories and not to show the exact positions of the various designs. No attempt should therefore be made to derive quantitative conclusions. Only with a more specific definition of degree of logic distribution and a more detailed analysis of the implementations involved can such positions be made more precise.

2.3 Logic-enhanced Secondary Storage Designs

2.3.1 Uni-Search-Processor Scheme

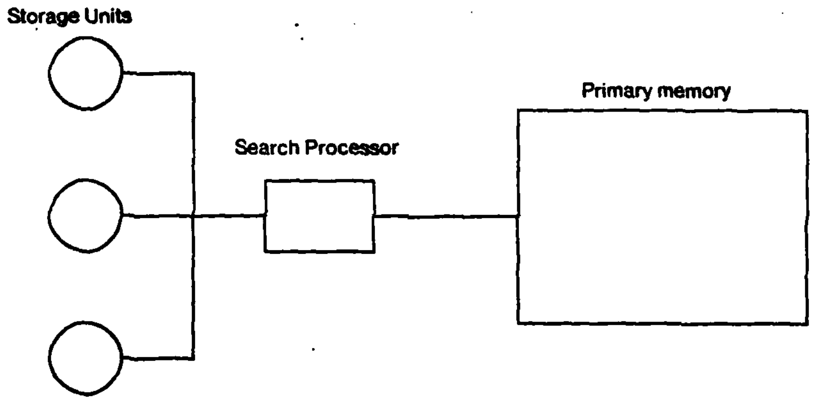


Figure 2-3: Uni-search-processor model.

One search processor is attached between the secondary storage devices and the primary memory (Figure 2-3). Irrelevant data can be filtered out before they reach the primary memory, and thereby reducing I/O traffic. Some early examples of database machines are of this kind. This scheme occupies the low end of the logic distribution spectrum. The *allocation of logic* is dynamic since one search processor serves the entire secondary memory. An example is the *Content Addressed File System*, or CAFS [3, 23].

2.3.2 Multi-Search-Processor Scheme - Static Allocation

A storage unit considered in the following is usually a disk track, bubble memory, or charge-coupled device. The static allocation scheme pre-allocates one search processor to each storage unit, requiring as many search processors as there are storage units (see Figure 2-4). With such designs, the search time is typically tens of milliseconds. They occupy the high end of the logic distribution spectrum.

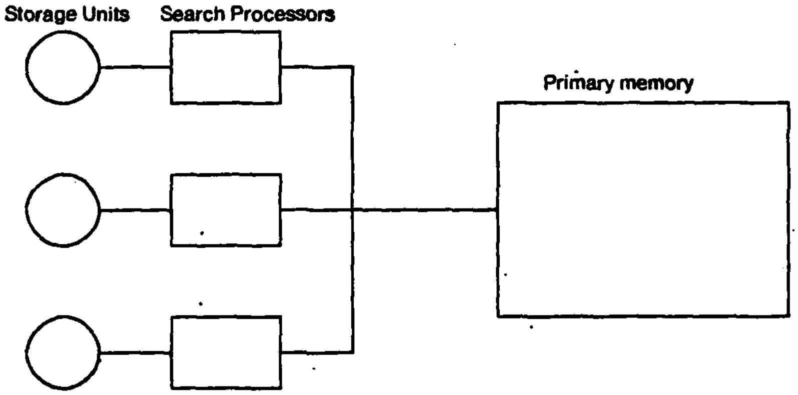


Figure 2-4: Multi-search-processor (static allocation).

Among the designs which fall into this category, we cite the following. (Some of these designs, conceived with static allocation in mind, are now shifting to the dynamic allocation scheme, to be discussed shortly afterwards.) CASSM [91], or Content Addressed Segment Sequential Memory, designed and with a prototype built at the University of Florida, is one of the earliest design efforts and thus has exercised considerable influence over other designs. It is designed mainly to support the hierarchical database model. The CASSM system consists of an array of cells each of which contains a processing element and a circular memory element (a disk track). The processing element consists of a read head, comparison and modify logic, and a write head. One difficulty of implementing logic-per-track devices is the mechanical tolerance problem [60]. As bit densities achievable on the magnetic media increases, the spacing between the read and write heads becomes tighter by the same amount. CASSM has a read/modify/write capability using two physical disk tracks (A and B) per logical track of storage. Data read from track A passes through the cell and is then written on track B. Next, data from track B is passed to track A, which permits any bit to be modified yet employs a single head per track. Data sensed by the read head can be compared against a constant value stored in the cell. CASSM has one comparator for data search. Intermediate results are recorded as mark bits which can subsequently be used. A hardware prototype with two cells was originally planned. What actually was built was a single-cell prototype system. A fixed-head floppy disk was used. The system was built using 220 SSI circuits and interfaced to a NOVA 800 computer.

RAP [79], or Relational Associative Processor, was designed and implemented at the University of Toronto. It supports the relational data model. Like CASSM, an RAP cell has a processing element and a rotating memory, which is a CCD track built from Intel's 16 Kbit 2416 components. One problem of using charge-coupled devices is their volatility. Since they are susceptible to power disturbances, appropriate measures (such as stand-by power sources) must be taken. In an RAP cell, three comparators are used. (Recall that a CASSM cell has one.) The comparators are used in two instances. The first instance is the qualification or selection of tuples. Up to three domains may be compared against constant values stored in the cell. Each tuple has also a mark bit field to record temporary results which can subsequently be used. A second instance is to compare single domain values against three comparands. It may thus require less number of revolutions as compared to CASSM. A prototype system, called RAP.2, of two cells was built in 1977. In this prototype system, each record was limited to 255 items whose length could only be 1, 2, or 4 bytes of data. In the planned RAP.3 this length will be arbitrary. In RAP.2 each cell can store data (or part of the data) belonging to a single relation. This means that while a large relation can be allocated to several cells, an entire cell is needed to store a small relation. This restriction will be removed in RAP.3. Regarding the performance of RAP.2, it is reported that gains range from one to three orders of magnitude in query execution speed over conventional systems.

Chang [17] proposes slightly modified major/minor loop bubble chips to accommodate storage and access for relational databases. While the data tracks of the logic-per-track disk files are difficult to synchronize, an important advantage of bubble-like devices is the bit-by-bit synchronization. Bubble technology, however, suffers from a bit rate problem, that is, the data transfer rate may be too slow to be competitive with other rotating storage devices.

RARES [68], also designed to support the relational model, differs from others mainly in that tuples are stored across the tracks of the head-per-track disk storage. A system of up to 14 search processors has been designed and implemented by Leilich et al. [63]. With considerably less logic distribution than the other designs in the static allocation category is the design by Lang et al. [59], in which a processor is associated with each direct access storage device, instead of with each storage track. A very recent effort is the investigation of tools for generating language transducers [32]. The research involves the construction and evaluation of VLSI circuits using language transducers for several problems. One of the problems is the construction of a data filter for logic-per-track database machines.

In the static allocation scheme, each storage unit has its own private search processor and it does not matter in which storage units a file should reside. Therefore storage organization is quite simple. The main problem, of course, is the waste of too much potential resource. The global database may contain information for many different users and applications. In processing one specific query, however, the actual load (i.e., the amount of data needed for the processing) may well constitute a tiny fraction of the whole database. Providing logic to all disk tracks of the entire database is analogous to a memory management system in which enough physical memory is provided to hold all programs ever written by all the users of a given installation.

2.3.3 Multi-Search-Processor Scheme - Dynamic Allocation

Still using our analogy with a memory management system, a virtual memory system allocates physical memory dynamically to segments of programs only when their presence in the main store is required for execution. The amount of physical memory can thus be significantly less than the total program space. Similarly, in the dynamic allocation scheme, a number of search processors are allocated dynamically to those storage units containing information to be processed. The amount of search logic is therefore distributed to the entire database and hence will occupy the lesser end of the logic distribution spectrum as compared to those with static allocation. Depending on how search processors are connected to storage units, we have many variations of this scheme. Two of these, which correspond to real database machine designs, will be examined.

2.3.3.1 Complete-Bipartite-Graph Connection

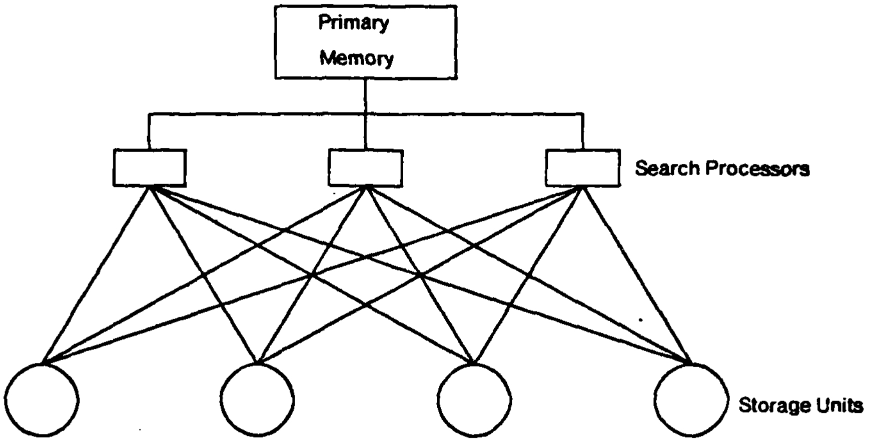


Figure 2-5: Complete-bipartite-graph connection.

In this scheme each search processor is connected to every storage unit of the database, as depicted in Figure 2-5. It works in the following manner. Each storage unit keeps broadcasting its contents to all the search processors. An individual search processor can choose to listen to one of the storage units and ignore the others. We have thus a very flexible connection. Any search processor can operate on any storage unit. Furthermore, several search processors can operate independently on the same storage unit, as long as they do not contend (for instance, they should not all try to update at the same time). This connection also allows a multi-user system in which some search processors may be operating to answer one user's queries, while some other search processors are working on another user's queries. Flexibility is of course obtained at the cost of the number of connections and the more complex control mechanism of the search processors. DeWitt [25], at the University of Wisconsin, Madison, proposes a design of a system called DIRECT which uses essentially this connection. In the proposed configuration, it will have eight LSI-11/03 search processors and 32 storage units each composed of a 16K byte CCD memory.

2.3.3.2 Partitioned-Storage-Units Connection

In this connection scheme, if we have t search processors and n storage units (typically, t ≪ n), then all the storage units will be divided into n/t partitions, each with t storage units. The t search processors can be connected to the storage units of one partition, but not to storage units of different partitions. Data residing in one partition can be examined by the search processors in essentially one disk revolution time (assuming a storage unit to be a disk track). Therefore, if related data are clustered into the same partition, they can be searched very quickly. We thus see that this scheme can provide the same performance as the static allocation scheme, given that we have enough search processors (t is sufficiently large) and related data are properly clustered.

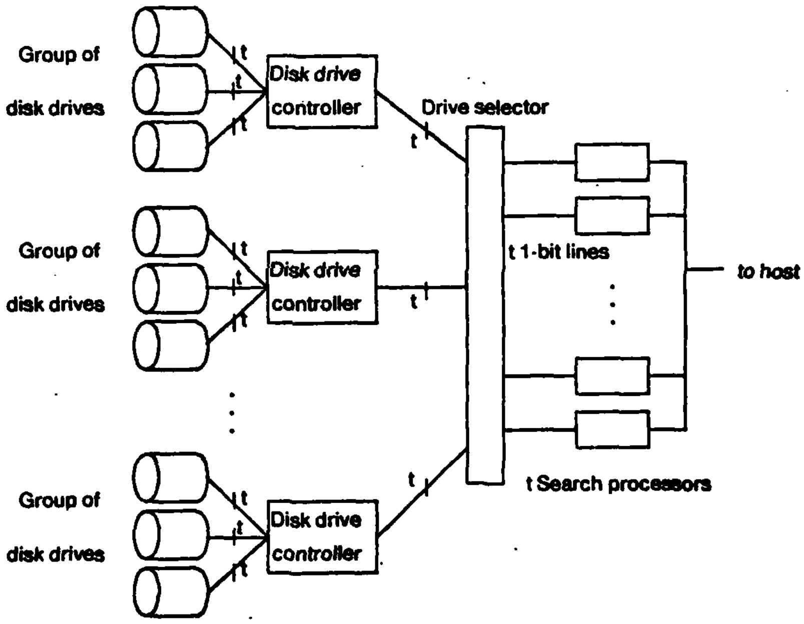


Figure 2-6: Partitioned-storage-units connection.

The Data Base Computer [5], or DBC, designed by a group headed by D. Hsiao at Ohio State University, fits into this model. The DBC design uses moving-head disks as storage devices, the only requirement being the parallel read-out capability of the t tracks of one disk cylinder. The set of disk drives is divided into 8 - 16 drives for access and control purposes

(Figure 2-6). Each group is controlled by a disk drive controller. A drive selector determines a particular disk drive controller which, in turn, determines the disk drive whose data are to be transferred. Data read out from one cylinder are then fed into t search processors. The DBC design was conceived with dynamic allocation approach in mind, and has provided extensive literature on such issues as data clustering and security checks [6].

2.3.4 Appraisal

Logic-enhanced secondary storage designs are based on the *logic-per-track* philosophy and have one common goal: that of providing efficient on-the-fly search of massive amounts of data in one or a few disk rotations. They constitute promising approaches to the important selection operation. Some other frequently used database operations, however, require not only knowledge of the values of individual data items, but depend on some kind of interaction among data items. The relational join between two relations of size n each, for example, requires $O(n^2)$ comparisons in its straightforward implementation. With a secondary associative storage device, it can be implemented as follows. For each tuple of one relation, we extract the specific field over which the join is being performed. Then in one revolution time we compare it with the corresponding field of all the tuples of the second relation. Therefore the join operation can in principle be obtained in approximately O(n) revolutions, where n is the number of tuples of the first relation. While this linear performance result might seem quite acceptable at a first sight, we have to keep in mind that one revolution time is on the order of tens of milliseconds. Therefore this mechanism is acceptable as long as we have a small number of argument tuples. For a concrete example, we cite the design of RAP in which the join is performed in a way very similar to the one described. Table 2-1 reproduces a summary of some results of a live demonstration of the RAP.2 prototype hardware [79]. The execution time for the third query, containing a join operation, is considerably greater than those of the other two.

Some recent designs combine the secondary associative storage devices with a logic-enhanced primary memory. In such designs, the secondary associative memory plays an important role in the case when the problem size is too large to be handled entirely in the primary memory. Appropriate partitions can be retrieved by the logic-enhanced secondary

Main operations	# rotations	Time	Tuples retrieved
select, average, sum	13 rotations	18/60 s	11 tuples
select, updates	5 rotations	8/60 s	0 tuples
select, join	63 rotations	82/60 s	7 tuples

Table 2-1: Summary of demonstration results.

storage devices and delivered to the primary store. For example, Lin [69] discusses the usage of associative secondary storage to aid in external sorting. Sorting is also discussed in the article on RARES [68]. The method is based on the knowledge of a histogram concerning the key values. By using content addressability of the secondary store, the appropriate partition is brought into the main memory which is assumed to be fast enough to produce sorted sequences in a pipelined fashion as a new partition is being retrieved. A more detailed discussion on these ideas is found at the end of Chapter 3.

2.4 Logic-enhanced Primary Storage Designs

Depending on the degree of logic distribution, several kinds of logic-enhanced primary storage designs can be considered. At the low end of the spectrum is the attachment of special-purpose hardware of limited size to a conventional passive memory. In such designs, logic is allocated dynamically to the entire memory. At the other end of the spectrum are the designs using the so-called smart memory (which we will refer to as *logic-per-datum* designs), in which there is a commingling of logic and memory elements in a fine grain. Such designs are of very high performance and constitute a departure from the von Neumann architecture. The remainder of this section will survey the various logic-enhanced primary storage designs.

2.4.1 The Post-Processors of The DBC Design

In the DBC design mentioned earlier, functions such as sorting of retrieved records, relational join operations on two sets of records retrieved from secondary memory, and the set functions as maxima and average, are all handled by what is known as the post-processors. Recall that search logic is allocated dynamically to the secondary memory so that a cylinder of data can be content searched in essentially one disk rotation. The retrieved data are fed, in a pipelined fashion, to the post-processors. The post-processing functions are presented in a number of reports describing the sort operation [40] and the join operation [39, 41]. In particular the last report also contains a comparison of the DBC join method with other proposed methods. The post-processing functions are performed by a multiprocessor system consisting of a number of linearly connected processors each with private memory. In an earlier description [39] they share an associative memory for storage and fast retrieval of join attribute values. For each unique join attribute value, the associative memory provides an integer index, which is then used by the processors to obtain a memory location by means of hashing. This first design has the drawback of not being easily hardware-extensible. All processors share the same associative memory which will become a bottleneck when the number of processors increases. A new design described in [41] distributes the associative memory among all the processors, such that each will contain a fraction of the original associative memory. In either design, a very strong assumption is that all the required tuples can fit into the memories of the multiprocessor system. Its practical use may depend on the ease of partitioning large problems.

2.4.2 The Hierarchical Associative Architecture

A hierarchical associative architecture has been proposed by Shaw [83] for the efficient evaluation of relational primitives such as join, project, and select. It consists of a hierarchy of associative storage devices under the control of a general-purpose processor. At the bottom of the hierarchy is a secondary associative memory (SAM), which may be implemented using parallel logic-per-track disks, as in CASSM, RAP or RARES. At the top of the hierarchy is a primary associative memory (PAM), capable of fast content-based searches. Complex relational primitives such as the join operation on two relations are evaluated in the primary associative memory, with the assistance of the secondary associative memory. Shaw considers the important case of handling large problems whose size exceeds that of the primary associative memory. He distinguishes two kinds of evaluations, namely, internal evaluation where the argument relations can be fit entirely into the primary associative memory, and external evaluation where the relations exceed its capacity. Shaw shows that when argument relations are large, the time required for evaluation of complex primitives with the hierarchical associative architecture represents a substantial improvement over the results attainable using only secondary associative storage devices.

2.4.3 Systolic Priority Queues

The systolic array approach has been proposed as a solution to offload costly computations. Systolic algorithms have been presented for matrix computations [51], signal and image processing [52, 53, 55], pattern matching [31], transitive closure and dynamic programming [49], and many others. (For a more complete list of systolic algorithms, see [54]; for a discussion on the philosophy of the systolic architecture, see [56].) One systolic design especially useful in database applications is the priority queue proposed by Kung and Leiserson [64]. A linear array of cells is used to store a collection of elements with the possible operations of insertion, deletion and minimum extraction. (Fisher [30] presents designs of systolic arrays for computing running order statistics where ranks other than the minimum and input spaces of higher dimensions are considered.) In addition to storage, some comparison logic is provided at each cell. A sequence of the above operations can be executed concurrently in a pipelined fashion, in such a way that the response time is a constant, independent of the length of the array. A priority queue can also be implemented with a hardware heap in a straightforward way as we will show in Section 3.3.3. Notice that a systolic priority queue, whether implemented as a linear array or a heap, is a logic-per-datum device and, as such, occupies the high end of the logic distribution spectrum. However, if such a device of limited size is used to aid in the internal sorting of a much larger collection of numbers, then the degree of logic distribution will be considerably less. Hence, depending on the size of a systolic device and the problem size it is able to handle, its usage may be economically infeasible, or perfectly viable and justifiable. (Internal sorting with the aid of systolic devices will be treated in more detail in Chapter 5.)

2.4.4 The Systolic Arrays for Relational Operators

Kung and Lehman [50] consider the use of a large number of simple processors connected in a linear array for the handling of relational operators. They describe, among others, arrays for performing intersection (which can also be used for projection with duplicate removal) and join of two relations. A single database transaction may consist of a number of relational operations. Therefore to process all the operations required in one or more transactions, an integrated system containing several systolic arrays is needed. A crossbar scheme connecting the memories holding required data and the special-purpose systolic arrays is proposed.

In a recent work by Kim, Kuck, and Gajski [43], a bit-serial/tuple-parallel relational query processor is proposed. The scope of the study is limited to designing a query processor that will efficiently process data already loaded into the primary memory. As in the case of the systolic arrays of Kung and Lehman, the proposed query processor is designed with the view toward VLSI implementation.

2.4.5 The Tree Machine

A logic-per-datum design consisting of a binary tree of cells has been proposed by Bentley and Kung [9] (and independently by Browning [14, 15]). The internal cells of the binary tree can propagate information to, as well as combine the information of the descendant cells (such as taking the logical and, or select the minimum, etc.). Data elements reside in the leaf cells which are provided with logic to carry out a limited repertoire of instructions. Such a structure is especially suitable for different kinds of searching problems, because of the logarithmic path between the root cell and any leaf cell. It has been extended [85] to handle the sort operation, and relational operations as project, join, union, etc. Details are found in Chapter 3. Such a design is of high performance and occupies the high end of the logic distribution spectrum (see Figure 2-2 under the branch of static allocation). The same device, of a small capacity, can be allocated to serve a much larger memory, thereby occupying the lesser end of the spectrum (Figure 2-2 under the branch of dynamic allocation). Such a device will be discussed in Chapter 5.

2.4.6 Appraisal

Logic-enhanced primary memory designs are useful for compute-bound tasks where a same datum participates in many operations. On the other hand, I/O-bound tasks such as selection are better handled by logic-enhanced secondary storage devices, before the data even get to the primary memory. The best architecture is perhaps a hierarchy containing both kinds of devices. Logic-enhanced secondary devices may be used to filter out the irrelevant data, and more complex operations on the selected ones are processed in the logic-enhanced primary memory. Designs where logic is allocated dynamically to the entire memory is usually economical to implement but require careful study of the issue of problem partitioning, that is, how to decompose a large problem such that it can be handled by a special-purpose device of smaller size. The logic-per-datum designs can provide very high performance and constitute a departure from the von Neumann architecture. Their implementation cost may however limit their usage to very specialized applications, where fast response time and throughput are required, e.g. on-line bank-teller systems that support a huge number of simultaneous transactions [62].

Chapter 3
The Tree Machine

In this chapter we describe a tree-structured machine for the handling of relational operations, as well as the important sorting operation. A novel space allocation scheme for the tree machine, as well as insertion and deletion procedures, will be described. Many of the sections presented here are contained in [85], which however does not consider problem partitioning.

In database applications, queries typically require the execution of a sequence of database operations before the answer is obtained. One such sequence might be the *selection* of the required tuples, the *join*ing of the selected tuples of two relations over some attribute, followed by the *projection* of the result relation to extract the desired columns and the final output of the result with *duplicate-removal*. It is therefore desirable to have a single special-purpose device which can provide efficient solution to all basic database operations. For this purpose we have chosen the tree machine of Bentley and Kung [9] and attempted to extend it to handle other basic database operations. Tree-structured machines have been proposed as general-purpose computing devices by Berkling [11], Browning [14, 15], Mago [72], Sequin, Despain, and Patterson [80] and Wilner [95]. Hollaar [35] presents a tree-structured design for merging sorted lists.

This chapter is organized as follows. First we characterize the system configuration and present a general description of the tree machine. We shall present a new space allocation scheme and comment on its practical usage. We then consider some basic database operations, including select, sort, project, join, union, and intersection. Finally we discuss decomposition strategies to retrieve to primary memory appropriate partitions of a large argument set stored in secondary memory.

3.1 System Configuration

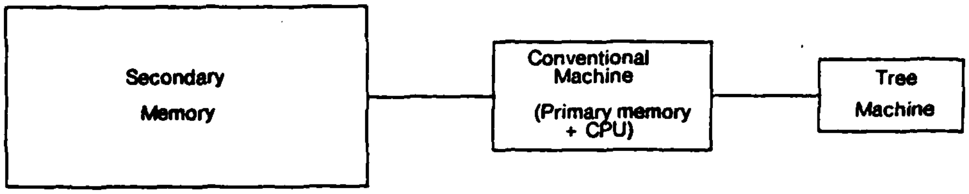


Figure 3-1: System configuration.

We distinguish the following hierarchy of memories (Figure 3-1): secondary memory, primary memory, and the special-purpose device. Under this hierarchy, two levels of problem decomposition need to be considered. On the first level a large problem exceeding the primary memory capacity has to be decomposed and appropriate partitions brought to the primary memory to be processed. Two decomposition schemes will be discussed at the end of this chapter. This level of decomposition can be carried out with the aid of some kind of secondary associative storage devices, as discussed in the previous chapter. A second level of decomposition is the partitioning of data elements inside the primary memory to be handled by a tree machine of smaller size. This will be discussed in Chapter 5. Throughout this chapter, however, we assume a "large" tree machine in the sense that every tuple stored in the primary memory has an alternate representation (such as its memory location and part of the complete tuple) stored in a node of the tree machine.

3.2 General Description of the Tree Machine

The tree machine has three kinds of nodes (Figure 3-2): O-nodes, □-nodes, and Δ-nodes. Each one of a collection of data elements resides in a □-node, which is provided with some logic to carry out a limited repertoire of instructions. The O-nodes broadcast streams of instructions and/or data to the □-nodes where they are executed in parallel. The □-nodes compute results which are then combined by the Δ-nodes to produce the final result. For example, selection of data satisfying a conjunction of conditions can be performed by broadcasting the conditions to the □-nodes which then decide which ones are to be selected. The Δ-nodes then take the selected results and output them through the root Δ-node. The

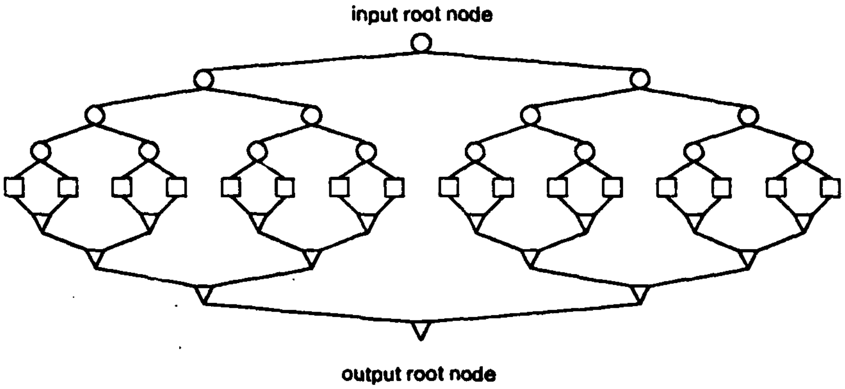


Figure 3-2: The tree machine.

structure of the tree machine is that of two complete binary trees, one being the mirror image of the other. In Chapter 4 we shall show that it can be nicely laid out on chips which, in turn, can be compactly laid out on circuit boards.

3.2.1 A New Space Allocation Scheme

We now review the insertion and deletion algorithms mentioned in [9], and propose a new space allocation scheme, along with the associated insertion and deletion procedures. Consider the task of maintaining a collection of elements in the □-nodes of the tree machine, with the possible operations of inserting new elements and deleting existing ones. One way of doing insertion is to maintain a count in each of the O-nodes, specifying the number of free □-nodes which are its descendants. Each time a new element is to be inserted, a O-node will pass on the element to the offspring which has free □-nodes below (choosing an arbitrary one if both are eligible). Then it will update its own count by decrementing it by one. Similarly, when an element contained in a □-node is deleted, some of the O-nodes need to have their counts updated. More specifically, these are all the O-nodes which lie on the path from the input root node to the particular □-node where deletion has occurred. This can be done by proceeding backwards from the deleted node to the input root node, adjusting the counts on the way up. O(log n) steps are therefore necessary to adjust the counts, where n is the total number of □-nodes of the tree. This scheme requires storage for the count, as well as the associated logic needed for its updates, in each of the O-nodes. Since counts need to be adjusted after a deletion, it makes pipelining an arbitrary sequence of insertions and deletions more difficult. Consider for example the tree with all the □-nodes already full, and the sequence of alternate deletes and inserts, as follows:

delete, insert, delete, insert, ...

After each deletion, O(log n) steps are needed to adjust the counts. The time to execute a sequence of m operations such as those above will be O(m log n).

We wish to design new insertion and deletion algorithms with the following two objectives:

- Arbitrary sequences of insertions and deletions can be easily pipelined.
- No counts or associated logic for their updates are to be maintained in the O-nodes.

We have found a way to achieve the above if the following assumptions are made:

- A single count is kept in the tree controller (the interface controller of the tree machine).
- For each delete command issued by the tree controller, there exists one and only one item in a □-node which will be deleted.

Maintaining one single count in the tree controller surely poses no problem, as compared to the n − 1 counts in the original scheme. The second restriction with respect to deletion will be discussed later.

What we are facing is a problem of dynamic allocation of free □-nodes. We wish to maintain a pool of such nodes. For each insert operation, we remove one free □-node from this pool; for each delete operation, we return the garbage node to the pool to be re-cycled. Consider each □-node as containing storage for two fields, Node.FreePosition and Node.Content. If a □-node is free, then Node.FreePosition contains an integer from 0 to n-1, where n is the total number of □-nodes in the tree. Also, for simplicity of notation, we write Node_i for the □-node whose FreePosition field contains i, $0 \le i \le n-1$. If a □-node is occupied then its FreePosition contains Λ. Node.Content is the value of the item stored in the □-node which, for simplicity, will be assumed to be an integer.

If the tree is empty (i.e., it stores the empty collection), we assume that the free □-nodes of

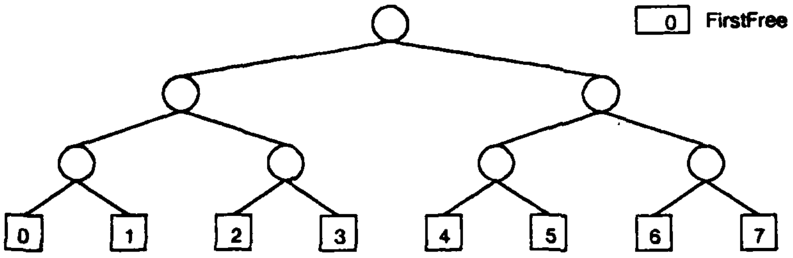


Figure 3-3: An empty tree.

the tree are Node$_0$, Node$_1$, Node$_2$, ..., Node$_{n-1}$, in any order. (See Figure 3-3, where the bottom half of the tree machine is omitted and only the FreePosition field is shown.) We also assume that the tree controller maintains an integer count called FirstFree, such that the free □-nodes are Node$_{FirstFree}$, *Node*$_{FirstFree+1}$, ... , *Node*$_{n-1}$. *FirstFree* contains 0 if the tree is empty and contains n if the tree is full.

3.2.1.1 Insertion

To insert an element X, the tree controller will generate an insert instruction which has two parts, namely, Instruction.FreePosition and Instruction.Content. Instruction.FreePosition will indicate which □-node is to be removed from the pool of free □-nodes. The tree controller assigns Node$_{FirstFree}$ to be that node. Instruction.Content contains the value to be inserted. This is shown as follows.

```
Instruction.FreePosition ← FirstFree;
Instruction.Content ← X;
FirstFree ← FirstFree + 1
```

Each O-node broadcasts the instruction to its two offspring. Simultaneously, each □-node will try to see if it has been selected as the node to receive the element being inserted. Exactly one such node will be found and this will mark itself as occupied after redefining its Content field. This is shown as follows. (Figure 3-4 shows the tree after 6 elements have been *inserted to an initially empty one.)*

```
if
  Node.FreePosition = Instruction.FreePosition
then
  Node.Content ← Instruction.Content;
  Node.FreePosition ← Λ
fi
```

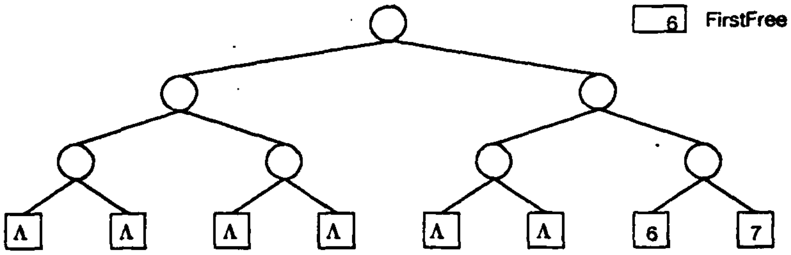


Figure 3-4: After six insertions (Λ denotes an occupied node).

3.2.1.2 Deletion

We consider deletion of an element from the tree based on the content of that element. Suppose that we wish to delete the element X from the tree. By the assumption made before, one and only one □-node will be freed whenever a delete command is issued. This means that the tree controller will know beforehand that one of the originally occupied □-nodes will be able to return to the pool of free □-nodes, even though it does not know which one. Therefore, the delete instruction issued by the controller will contain not only the content X to guide the deletion, but also the value that should be stored into the FreePosition field of the node to be freed.

```
FirstFree ← FirstFree - 1;
Instruction.FreePosition ← FirstFree;
Instruction.Content ← X
```

Again the O-nodes merely broadcast the delete instruction to the □-nodes. Each □-node will attempt to match its content with that in the instruction. Only one will find a match and that one will be immediately returned to the free pool by redefining its FreePosition field. This is shown as follows. (Figure 3-5 shows what remains after two deletions have been made to the example illustrated by Figure 3-4.)

```
if
  Node.Content = Instruction.Content
then
  Node.FreePosition ← Instruction.FreePosition;
  Node.Content ← Λ
fi
```

Here we have used Λ to indicate the null content. Note that the functions performed by the □-node in the insert and delete commands are symmetrical. We obtain one from the other by merely interchanging the words *Content* and *FreePosition*. Based on this, we now give a simpler version of these algorithms.

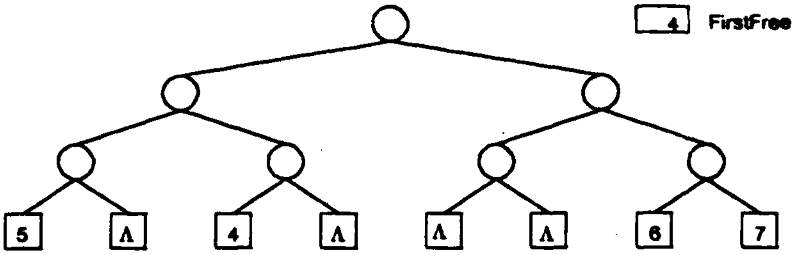


Figure 3-5: After two deletions (Λ denotes an occupied node).

3.2.1.3 A Simpler Version

Notice that Node.FreePosition and Node.Content are never simultaneously defined; whenever one is defined, the other is Λ. Therefore they can occupy the same storage in a □-node, which we now call Node.Storage (Figure 3-6). If the first bit of Node.Storage is 0 or 1, then the remaining bits contain a *FreePosition* or a *Content*; respectively.

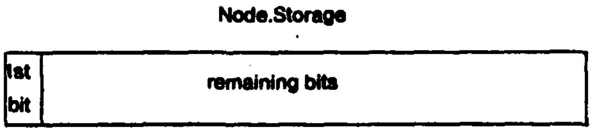


Figure 3-6: A node storage.

The insert and delete instructions now have a unique format, as shown in Figure 3-7. The left bit indicates whether the remaining bits of the left part is a *FreePosition* or a *Content*, using the same convention established before. Similarly, the same applies to the right bit relative to the right part. In an insert instruction, we define the left bit = 0 and the right bit = 1; in a delete instruction, we define the left bit = 1 and the right bit = 0. In fact, only one of these two bits is needed, since one is always the complement of the other.

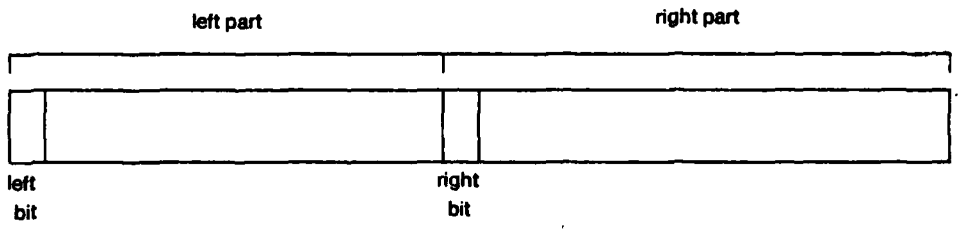


Figure 3-7: Insert and delete instruction formats.

The interesting thing to observe is that now the □-node does not have to distinguish an insertion from a deletion, since it will always proceed in the same way in both cases, as follows.

```
if
  Node.Storage = left part
then
  Node.Storage ← right part
fi
```

We have simplified the above presentation by assuming the item residing in a □-node to be an integer. The proposed algorithms can readily be adapted to handle more general cases with minor modifications.

3.2.1.4 Comments on the Algorithms

In the original insertion scheme mentioned at the beginning of this section, the element to be inserted is passed down the tree through a path of O-nodes until a free □-node is reached. The selection of this path is guided by the O-nodes which use their own count information as well as those of their two offspring. In the new scheme, the element being inserted does not follow any particular path, but is merely broadcast to all the □-nodes. It takes advantage of the content addressability of the tree machine to do the selection of the free □-node. Also, in the original deletion scheme, log n counts in the O-nodes need to be adjusted. Since we do not know which counts are to be incremented until the deletion is done, pipelining was not so easy to achieve. Here we have only two values to be adjusted, namely, those of FirstFree and of the FreePosition field of the deleted node. The interesting thing is that both values can be determined at the time the delete command is issued by the tree controller. Pipelining arbitrary sequences of deletes and inserts presents no problem at all. In some sense, we have *factored out* the counts and logic from the O-nodes to the tree controller, thereby reducing the space needed for its implementation. In VLSI designs, there is often a trade-off between space and time. In this case, however, the new space allocation scheme has allowed us to reduce space requirements and at the same time achieve better performance. We have, however, traded off generality for these improvements. A discussion of the restrictions of the algorithms now follows.

3.2.1.5 Comments on the Restrictions

The scheme is based on the assumption that, for each delete command, there is one and only one element qualified for deletion. Two situations may violate this assumption. In the following we discuss each case as well as the problem it causes.

- A delete command finds many elements to be deleted. (Notice this is never the case if deletion is based on a primary key, i.e., a field with values that uniquely identify the elements of the collection.) One delete command may cause more than one □-node to be deleted and, consequently, all such nodes will have the same value for their FreePosition fields. As a result, the next insert operation will insert the same information at more than one □-node. Depending on the type of transactions we are working on, this may cause no damage at all, but some waste of space. Therefore one solution may be perhaps to do nothing to prevent its occurrence, and perform a remove-duplicates operation at the moment the data elements are unloaded from the device. (Three solutions for removing duplicates are given in Section 3.3.4.)

- A delete command finds no element qualified for deletion. This may cause a more serious problem. The tree controller, in issuing a delete command, foresees the freeing of an occupied □-node and supplies a *FreePosition* for this purpose. A subsequent insert operation may be issued carrying this *FreePosition* expecting to find the free □-node where insertion is to occur. Instead, no such free node exists and the failure of the insert operation may cause erroneous results in a subsequent operation, say a membership testing or deletion operation. One solution is to make insert and delete work at different "ends" of the free pool. This of course will complicate the function of the tree controller.

A general and straightforward solution requires some additional processing before a delete command is issued. First those elements qualified for deletion are selected and deletion proceeds by using fields which uniquely identify them (such as a primary key, or an absolute address of the □-node). With this, we lose the pipelining advantage offered by the new scheme which, nevertheless, is still interesting because less storage and logic are required in the O-nodes. There are many examples of applications in which deletion is performed on previously inserted records. In an inventory control application, a record indicating an order is inserted and later needs to be removed when the ordered merchandise arrives. In a design database where pairwise intersections of rectangles are to be reported (Section 6.3), a record is inserted when a scan line touches the left edge of a rectangle, and subsequently has to be removed when the corresponding right edge is scanned. In such cases, the presence of the record to be deleted is certain. Furthermore, if deletion is based on a primary key, then the assumptions in question are met and the proposed scheme can be applied to its full advantage.

3.2.2 Disciplining the Data Flow

In operations where only one output is involved, new commands can be issued to the tree machine while the results are being handled at the Δ-nodes to be output. In other words, pipelining is easily achieved. In some operations, however, many results are produced in the □-nodes. These will traverse through the Δ-nodes until they reach the output root node. Given the funneling nature of the output binary tree (i.e., the bottom part of the tree machine), the Δ-nodes must cooperate among themselves in order to produce an orderly evacuation of the many results. We say that a Δ-node is ready to accept data if its storage is empty. It will then examine its two offspring and take the contents of a non-empty one. If both offspring have information to be transmitted, then it will select one according to some fixed rule (such as always picking the leftmost, or selecting the one with minimum value on some specified key). Another form of disciplining the data flow in the Δ-nodes involves the exchange of information between a Δ-node and its offspring Δ-nodes, to be described in Section 3.3.3.

Multiple results produced in □-nodes may have to be retained for a while before they are accepted by the Δ-nodes. In order to protect these results from being destroyed by the incoming stream of instructions or data, the broadcasting of information in the O-nodes cannot be done in a lock-step fashion. Rather, some request/acknowledge signaling convention is used. Each O-node ready to accept data sends a request signal to its parent node and goes into a wait state. Each O-node with data to be broadcast sends the data and an acknowledge signal after it has received the requests from both offspring nodes. Section 4.2.1 contains more details. With regard to such rules, the following observations can be made.

3.2.2.1 Observation 1

If any result formed in a □-node is always readily taken out without delay, then broadcasting items a_1, a_2, a_3, ... down the tree will result in alternate empty layers of O-nodes. With a tree machine of n □-nodes, (log n)/2 layers of O-nodes will be empty, as shown in Figure 3-8. Also, it takes log n steps for any item a_i (counted from the instant it enters the input root node) to reach a □-node.

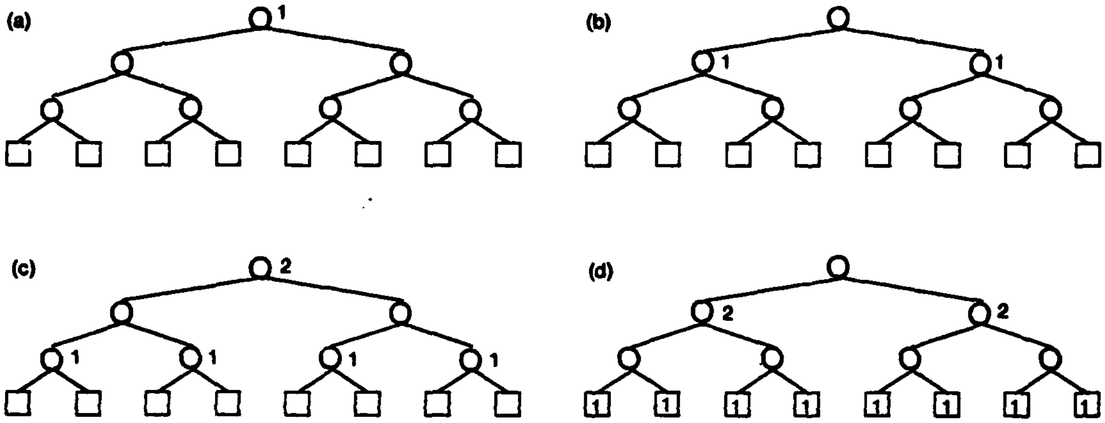


Figure 3-8: Alternate empty layers of O-nodes.

3.2.2.2 Observation 2

Consider a situation as above, in which alternate layers of O-nodes are empty. Suppose now the result computed in some of the □-nodes cannot be removed by the Δ-nodes for some period of time. This □-node will therefore start to block the flow of information above it until all the O-nodes on the path leading to the input root node are filled (see Figure 3-9). Since alternate layers of O-nodes are originally empty, (log n)/2 more new elements can still enter the tree before the path in question becomes full. Each of these new elements enter the input root node every other step. Therefore it takes log n steps to fill up this path.

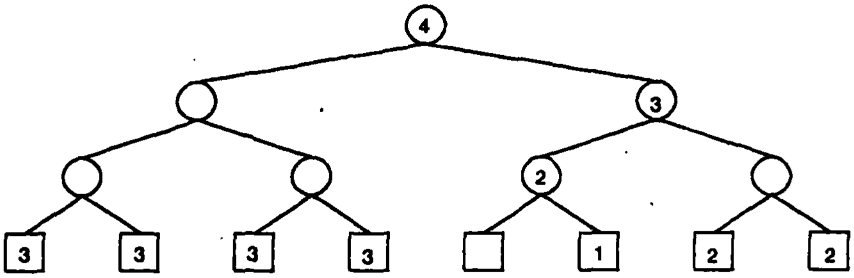


Figure 3-9: Blocking of flow.

3.2.2.3 Observation 3

If at a certain instant all the □-nodes are empty (creation of an "empty layer"), then this "empty layer" will be propagated toward the top of the tree in log n time. (Also, if the creation of an "empty layer" of □-nodes occurs every other step in a total of log n steps, then (log n)/2 alternate empty layers of O-nodes will be created, as indicated in Observation 1.)

3.2.2.4 Observation 4

Since each O-node broadcasts to its two offspring only if both are ready, any item a_i which enters the tree will reach all the □-nodes, though not necessarily at the same time. However, the items of a given sequence will visit a fixed □-node in the *same* order they entered the input root node.

3.3 Database Operations

First we briefly mention the selection operation. Next we present a sorting method which can be viewed as a parallel two-way straight merge sort. Then we discuss projection, as well as the join, union, and intersection operations.

3.3.1 Notation and Assumptions

Relations will be denoted by A, B, C, ... The cardinality of relation X is denoted by |X|. The number of □-nodes in the tree machine is denoted by n. Relational operations are applied on argument relations and produce results which are also relations. We use R to denote the result relation. Depending on the number of argument relations we consider two types of operations. Operations such as select and project require one argument relation A, while operations such as join and union are performed on two argument relations A and B. In either case, we will assume that the needed argument tuples can fit into the primary memory. Furthermore, we assume that the location and some attribute values corresponding to each tuple of A can be stored in a □-node of the tree machine.

3.3.2 Select

Selection can best be done in the logic-enhanced secondary storage devices, and only selected data will be retrieved to the database machine. However, the ability of performing the selection operation inside the tree machine may be useful when it is to be carried out on intermediate results. A variety of search problems has been considered in [9]. For example, suppose that each □-node contains the year of birth, salary, and an employee id number. Consider the range search query "retrieve all employees born between 1930 and 1950, whose salary is in the range [15000, 24000]". The appropriate instructions and constants (such as 1930, 1950, ...) are then broadcast to all the □-nodes where comparisons take place and output fields are set. The selected employee ids can then be output through the tree of Δ-nodes. (An instruction set will be proposed in Chapter 4.)

3.3.3 Sort

We assume all the data to be sorted reside in the □-nodes. Sorting with a tree machine of "small" size will be discussed in Chapter 5. As described earlier in the section on disciplining of data flow, each Δ-node ready to take data will select one non-empty offspring to accept its information. The following rule also achieves the same effect. Each Δ-node of odd and even levels alternately executes the step: "Examine its own data and those of its two offspring, rearranging them if necessary such that it will contain the minimum on a given key." Let (Location, Key) be the information contained in a Δ-node executing this step, where Location denotes the memory location of the data of which Key is a part. (The need for Location will become clear in the lexicographic sort to be seen in Section 3.3.3.1 and in the sorting algorithms to be presented in Chapter 5.) Let (Location$_1$, Key$_1$) and (Location$_2$, Key$_2$) denote the corresponding information in its left and right offspring, respectively. This step can be formulated as follows.

```
MinKey ← minimum(Key$_1$, Key$_2$);
if
  MinKey < Key
then
  if
    Key$_1$ = MinKey
  then
    (Location, Key) ↔ (Location$_1$, Key$_1$)
  else
    (Location, Key) ↔ (Location$_2$, Key$_2$)
  fi
fi
```

All Δ-nodes are initialized to contain (dummy location, ∞). A node containing ∞ key will therefore correspond to a node ready to take data. By exchanging its infinity key with an offspring holding valid data, it frees this offspring and enables it to accept data. The need for Δ-nodes of odd and even levels to operate alternately arises if there is possibility of a node having a key value greater than that contained in its offspring but less than its parent's.

Consider sorting the tuples of relation A by Key. For each tuple of A we keep a pair (Location, Key) in a □-node of the tree machine. After O(log n) steps, the bottom half of the tree (formed by Δ-nodes) becomes a heap-like structure, where the key value at each parent node is either infinity or less or equal to those of its offspring. The pair (Location, minimum key) is available at the root Δ-node and can be replaced by (dummy location, ∞). With this, the heap property may be temporarily lost. The actions of the Δ-nodes, however, will attempt to restore it. The next element of the sorted list will be available in the next step, and it too can be output and replaced by (dummy location, ∞). This goes on until the desired sorted list is obtained. Therefore the sorting time is O(|A|) + O(log n), dominated by the time to do input and output between the host and the tree machine.

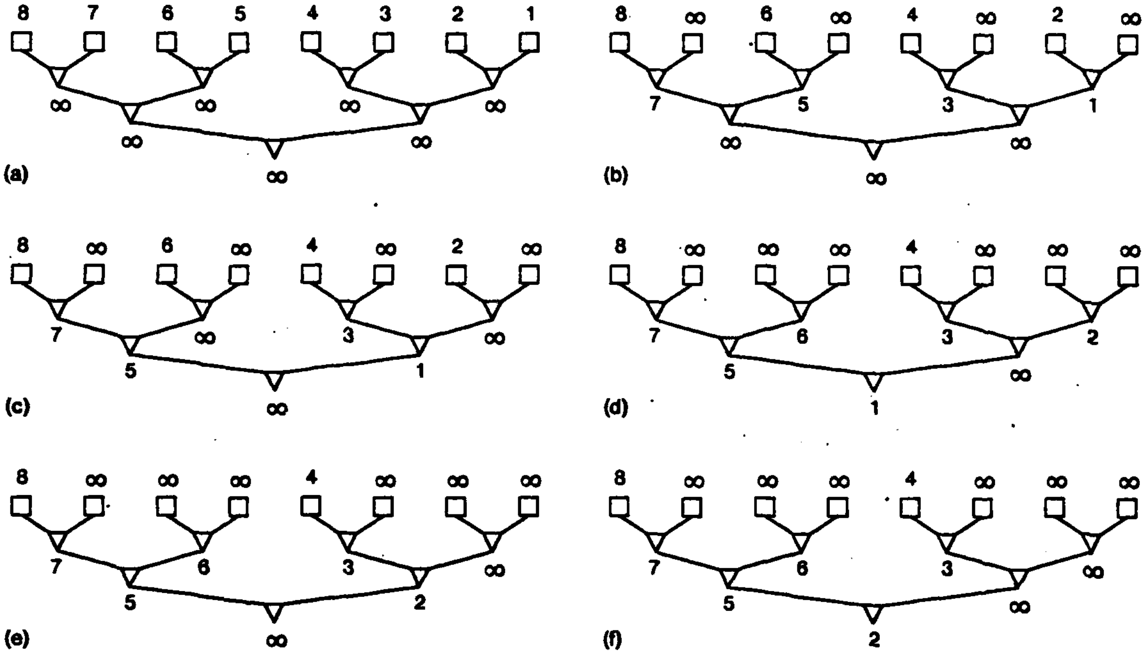


Figure 3-10: A sorting example.

The first steps of a sorting example are shown in Figure 3-10, where only the bottom part of the tree machine is represented. Only key values are shown and these can reside in arbitrary positions of the □-nodes. This sorting algorithm can be viewed as a parallel two-way straight merge sort. With n elements to be sorted in the □-nodes, n/2 two-element ascending runs are produced by the first-level Δ-nodes which are parents of the □-nodes. The second-level Δ-nodes produce n/4 four-element runs, and so on. Notice that an ith-level Δ-node needs storage for one element rather than 2^i elements. The ordering of each run is implicit in the relative positions of Δ-nodes its elements occupy.

3.3.3.1 Handling Long Key Fields by Lexicographic Sort

Consider the problem of sorting a sequence of m tuples of k component integers each into lexicographic order. Lexicographic order is, for example, the order of words in a dictionary, if the strings of letters are taken as tuples. If the component integer length corresponds to the maximum key size the sorting device can handle, solution of the above problem allows us to sort long key fields exceeding that length. (Refer to [65] for a work on the handling of variable key field lengths.)

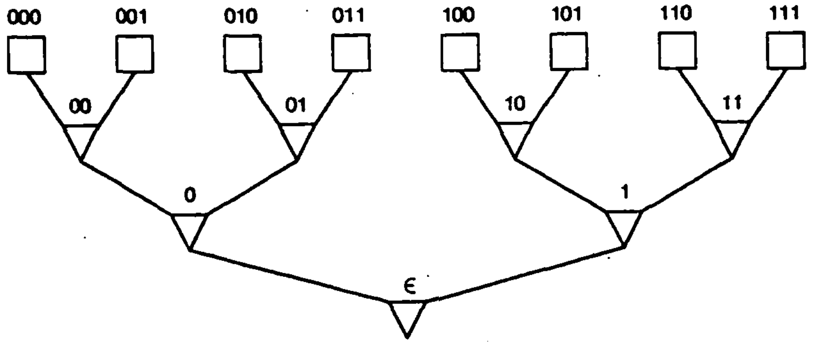


Figure 3-11: Labeling the nodes of a binary tree.

Let the k-tuples to be sorted lexicographically be $(K_{i1}, K_{i2}, \ldots, K_{ik})$, for $i = 1, 2, \ldots, m$, and let the memory locations of the tuples, Location_i, be contained in an array called FIFO. A total of k passes will be used, and in each pass one of the component integers is sorted. At the start of each pass, the FreePosition fields of the □-nodes "from left to right" have successively increasing values from 0 to n-1 (as in Figure 3-3). This can be stated in a more formal way by the following labeling definition for the nodes of a binary tree. (See Figure 3-11). Let the label of the root be the empty string ε. If the label of a node is λ, then its left and right offspring have labels λ0 and λ1 (concatenation of λ with 0 and 1), respectively. The labels of the □-nodes will correspond to their FreePosition values at the start of each pass. On the jth pass, each $(\text{Location}_i, K_{i,k-j+1})$ is inserted into the tree machine in the same order Location_i appears in FIFO. Since □-nodes of increasingly higher FreePosition values are assigned to the elements being inserted, the ordering in FIFO is retained in the relative position of □-nodes. This requirement is essential in the lexicographic sort because we do not want to lose the partial ordering already obtained in previous passes. By the way the sorting step is defined in the last section, if a Δ-node finds equal keys in both offspring nodes, the left one will be considered for exchange. This can be viewed as if a key K in a node of label λ has been extended to Kλ, with λ representing the less significant part. As the sorted components emerge from the device, their ordering is used to update that of FIFO. After p passes, the tuples in FIFO will be lexicographically sorted according to their rightmost p components. Thus after the kth pass, FIFO will contain the locations of the lexicographically sorted tuples. The time for the lexicographic sort is proportional to m k, or the product of the number of tuples and the number of components in each tuple.

3.3.4 Project

Assume again that relation A resides in primary memory. The main problem of projecting A over a specified compound attribute is the subsequent removal of duplicates. Three solutions will be described.

3.3.4.1 Solution 1

Solution 1 is a general solution that can take care of the case in which the compound attribute length exceeds the key length the sorting device is capable of handling. It consists of simply sorting the |A| tuples on the compound attribute using the lexicographic sort algorithm already seen. By examining the sorted list, duplicates can be detected and removed. This solution requires at least |A| □-nodes to hold the |A| input pairs (location, one component key). The time complexity is linear in the cardinality of relation A times the number of component keys.

3.3.4.2 Solution 2

This solution is interesting if the |A| pairs (location, compound attribute value) are already inside the □-nodes (e.g., they are results from previous computations in the tree) and the compound attribute length is short enough to be handled by the sorting device. All the Δ-nodes participate in the duplicate-elimination process, as follows. Each Δ-node compares its own compound attribute value with the minimum value of its two offspring. If its own value is greater than this minimum, it exchanges its data with that of the offspring nodes holding the minimum.

```
(TempLocation, TempKey)←(Location, Key);
MinKey←minimum(Key$_1$, Key$_2$);
if
  MinKey < Key
then
  if
    Key$_1$ = MinKey
  then
    (Location, Key)←(Location$_1$, Key$_1$);
    (Location$_1$, Key$_1$)←(TempLocation, TempKey)
  fi;
  if
    Key$_2$ = MinKey
  then
    (Location, Key)←(Location$_2$, Key$_2$);
    (Location$_2$, Key$_2$)←(TempLocation, TempKey)
  fi
fi
```

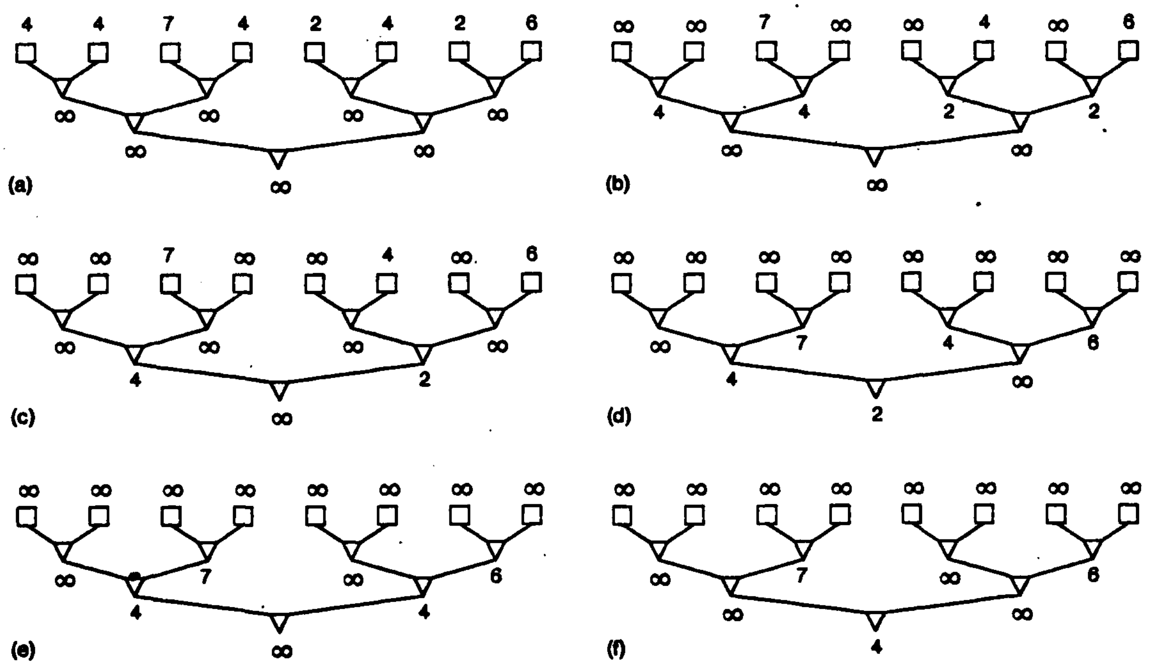


Figure 3-12: Duplicates elimination.

The difference with the previous sorting algorithm is that here the node exchanges its data with *both* offspring if they both hold equal key values less than its own. In the sorting algorithm, only one such exchange is performed. Figure 3-12 illustrates an example. Again this solution requires $|A|$ □-nodes, one corresponding to each argument tuple. The time complexity, however, is now linear in the cardinality of the result relation R. Depending on the amount of redundancy, this may represent a considerable improvement.

3.3.4.3 Solution 3

This solution is applicable if each compound attribute value is short enough to fit into a □-node. Duplicates can be detected and eliminated while the tree is being loaded. For each compound attribute value (denoted by Key), the tree controller issues two commands: a membership command to test the presence of Key in the tree, followed by an insert (NodePosition, Key) command where NodePosition specifies the node where insertion is to occur (cf. Section 3.2.1.1). Since the membership test result will be available only after 2 log n steps, the tree controller will have to have some mechanism of associating the membership testing result with the particular node position where duplicate key occurs. For example, a FIFO-type structure can be used. If the answer to the membership command is negative, the insertion of Key is indeed necessary. In case the tree controller gets a positive answer, it will locate the corresponding node position and "delete" that node by allowing the next insert command to overwrite this node. Denoting by NodePosition' the node position which holds a duplicate value, we have the following.

```
Obtain the next Key to be inserted;
Consider the answer to a previously issued member command;
if
  member gives positive answer
then
  Obtain NodePosition' from FIFO;
  NodePosition←NodePosition';
else
  NodePosition←FirstFree;
  FirstFree←FirstFree - 1
fi;
Issue command member(Key);
Issue command insert(NodePosition, Key);
Update FIFO with NodePosition
```

Notice that at any given instant the □-nodes may contain duplicated values, but the total number of such nodes will not exceed 2 log n. After all the tuples of the argument relation A have been considered, the tree controller may still receive answers of member commands reporting duplicates. In this case, it will simply issue dummy insert commands to overwrite such redundant nodes. The dominant time is linear in the cardinality of the argument relation A. The nice thing about this more complicated solution is that the number of □-nodes required is now equal to the cardinality of the result relation R plus 2 log n.

3.3.5 Join

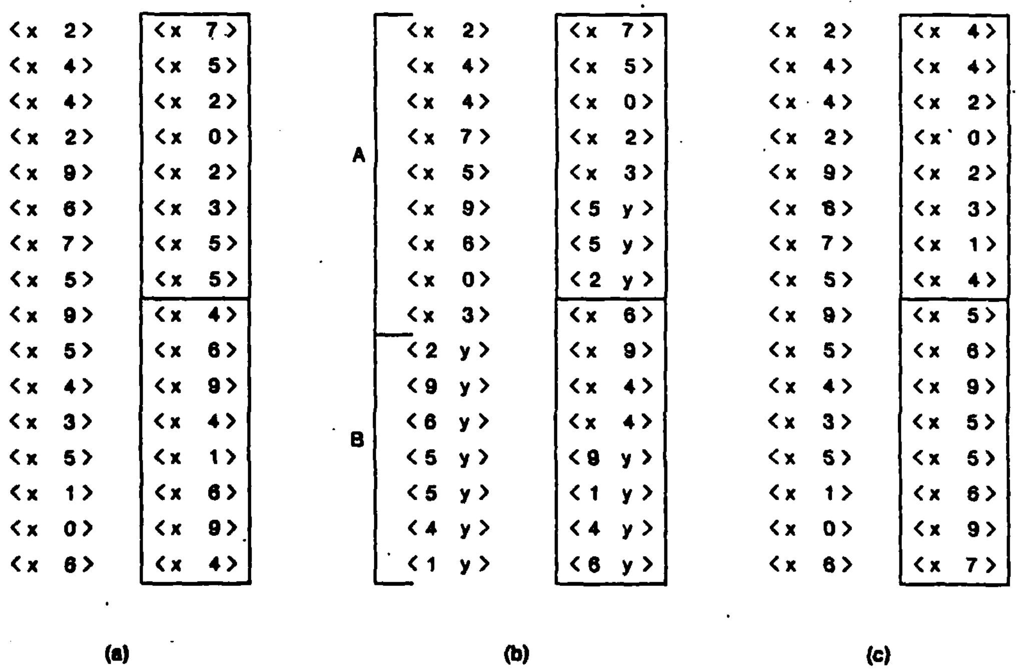
A detailed study of ten different join methods under various assumptions has been made by Blasgen and Eswaran [13]. Its findings were used as a basis for the join techniques used in System R [2]. Among the two methods that were selected as the best, one consists of the sorting of the two argument relations by their join fields and the subsequent merge where values are matched. Having already described the sorting method in previous sections, we proceed to describe another possible solution. All the tuples of relation A are kept in primary memory, with pairs consisting of their memory locations and join attribute values, (Location$_A$, Attribute$_A$), stored in the □-nodes. For each tuple of relation B, pairs consisting of (Location$_B$, Attribute$_B$) are broadcast to the □-nodes where the two join attribute values are compared. The matching cases will output the two corresponding locations so that the tuples can be located and concatenated to form a result tuple. The primary memory requirement is to hold |A| A-tuples and log |B| B-tuples. The time to perform the join is proportional to |A| + |R|, where R is the result relation.

3.3.6 Union and Intersection

Again, we can first sort the two relations and then obtain the result relation by simply doing a scan through both sorted sequences. Since whole tuples need to be sorted, the lexicographic sort probably will be necessary. If a tuple can fit into a □-node, a second solution to obtain the intersection is similar to the removal of duplicates, with the exception that no duplicates can occur within each argument relation. If we store relation A in the □-nodes, then a sequence of member queries for tuples of relation B suffices to produce the result. The union operation can be carried out similarly.

3.4 Partitioning Strategies

The algorithms we have presented depend on two strong assumptions (Section 3.3.1). The first assumption is that all the tuples needed for the evaluation of the sort operation or relational operation can fit into the primary memory. The second one requires a large special-purpose device which always meets the size requirement. For such algorithms to be of any practical significance at all, provisions must be made to handle cases in which these assumptions do not hold. In this section we assume that argument relations reside on secondary storage and discuss how appropriate partitions that fit into the primary memory can be selected and brought into the memory. This will be called external evaluation. (Much of the nomenclature used in this section is taken from [83]). Once a partition of data is inside the memory, their handling by a smaller special-purpose device will be discussed in Chapter 5.



<x 2>	<x 7>		<x 2>	<x 7>	<x 2>	<x 4>
<x 4>	<x 5>		<x 4>	<x 5>	<x 4>	<x 4>
<x 4>	<x 2>		<x 4>	<x 0>	<x 4>	<x 2>
<x 2>	<x 0>		<x 7>	<x 2>	<x 2>	<x 0>
<x 9>	<x 2>	A	<x 5>	<x 3>	<x 9>	<x 2>
<x 6>	<x 3>		<x 9>	<5 y>	<x 6>	<x 3>
<x 7>	<x 5>		<x 6>	<5 y>	<x 7>	<x 1>
<x 5>	<x 5>		<x 0>	<2 y>	<x 5>	<x 4>
<x 9>	<x 4>		<x 3>	<x 6>	<x 9>	<x 5>
<x 5>	<x 6>		<2 y>	<x 9>	<x 5>	<x 6>
<x 4>	<x 9>		<9 y>	<x 4>	<x 4>	<x 9>
<x 3>	<x 4>		<6 y>	<x 4>	<x 3>	<x 5>
<x 5>	<x 1>	B	<5 y>	<9 y>	<x 5>	<x 5>
<x 1>	<x 6>		<5 y>	<1 y>	<x 1>	<x 6>
<x 0>	<x 9>		<4 y>	<4 y>	<x 0>	<x 9>
<x 6>	<x 4>		<1 y>	<6 y>	<x 6>	<x 7>

(a) (b) (c)

Figure 3-13: Examples of key-disjoint partitions

The main idea is to decompose argument tuples into partitions in such a way that each particular tuple needs to be input only once. This means that *all the tuples which possibly* might interact should belong to the same partition. In the case of project where the main difficulty is the removal of duplicates, the partitions should be *key-disjoint* in the sense that no tuple in a partition can have a key (the projected compound attribute) equal to that of a tuple in different partition. In other words, all the duplicates, if any, belong to the same partition. Figure 3-13 (a) illustrates an example where only the projected attribute is explicitly shown, with the remaining columns represented by x. In the first partition shown, the duplicate values are 2 and 5, and in the second partition the duplicates are 4, 6, and 9.

For a two-argument operation such as join (similarly, union or intersection), a key-disjoint partition contains tuples of both argument relations with no tuple in one partition having a key (the join attribute) equal to that of a tuple in a different partition. This means that if two tuples, one from each argument relation, are to be concatenated to form a result tuple of the join, they should belong to the same partition. Figure 3-13 (b) depicts an example where only the join fields of the argument relations A and B are explicitly shown.

In the case of sorting, the keys of tuples in a partition should also be contiguous so that each partition being input to the primary memory contains keys greater than those of partitions already input. An example is illustrated in Figure 3-13 (c).

It should be clear that the partitions thus defined are not guaranteed to fit into the primary memory and a tuple may in fact have to be input more than once. In the worst case, all the keys of all the argument tuples may be equal, resulting in one single partition which is the whole argument set. Details of overflow handling can be found in [83].

Two decomposition procedures to select the desired partitions will be discussed. The first one is the hashing method due to Shaw [83]. It is suitable for handling external evaluation of project, join, union, and intersection. The second one is a statistical method based on ideas used in an internal sorting algorithm, due to Weide [92]. We show that these ideas can be used in combination with the logic-per-track concept to produce a partitioning method for join, project, and external sorting.

3.4.1 Partitioning by Hashing

Let t denote the total number of argument tuples and c the capacity in tuples of the primary memory. If the keys are distributed uniformly over some range $[x_{min}, x_{max}]$, then this range can simply be divided equally into $p = t/c$ subranges each of which will be used to delimit the range of one partition. Successive partitions can then be read into the memory in a monotonic sequence of the key ranges. The uniform distribution assumption is of course too strong. To handle non-uniform distributions, a hash function H can be used to scramble the key values so that, given a large set of keys, we hope to obtain a uniform distribution of the hashed values. In other words, if h_{min} and h_{max} denote the *minimum* and *maximum* of the hashed values, then we divide the range $[h_{min}, h_{max}]$ into p subranges each of which will determine one partition. The value of p is given by the expression

$$p = \lceil (1 + W)\, t/c \rceil,$$

where W is a waste factor. Shaw [83] shows that, if the distribution of hashed values is indeed close to uniform, then a relatively modest value of W (around 0.1) should suffice to make the cost of overflow recovery negligible. Success of the partitioning process by hashing thus relies on the randomizing property of the hash function. The distribution of hash values may *not always be close to uniform.* Notice that this method does not produce partitions with contiguous keys and therefore is not applicable for external sorting.

3.4.2 Partitioning by Using the Distribution Function

The distribution function F_X of a random variable X is a function such that, for each real x,

$$F_X(x) = P[X \leq x],$$

or the probability of the event $X \leq x$. The function F_X is nondecreasing with the following property.

$$\text{If } x_1 < x_2, \text{ then } P[x_1 < X \leq x_2] = F_X(x_2) - F_X(x_1).$$

If the key values x are uniformly distributed, as shown by their distribution function $F_X(x)$ in Figure 3-14 (a), then equally spaced *divisions on* $[x_{min}, x_{max}]$ *will* determine equally spaced divisions on [0, 1] along the $F_X(x)$-axis. Thus if each of the equal subintervals on $[x_{min}, x_{max}]$ is used to determine a subrange of a partition then, by the above property, the size of each partition will be about the same. However, if the keys are not uniformly distributed, as

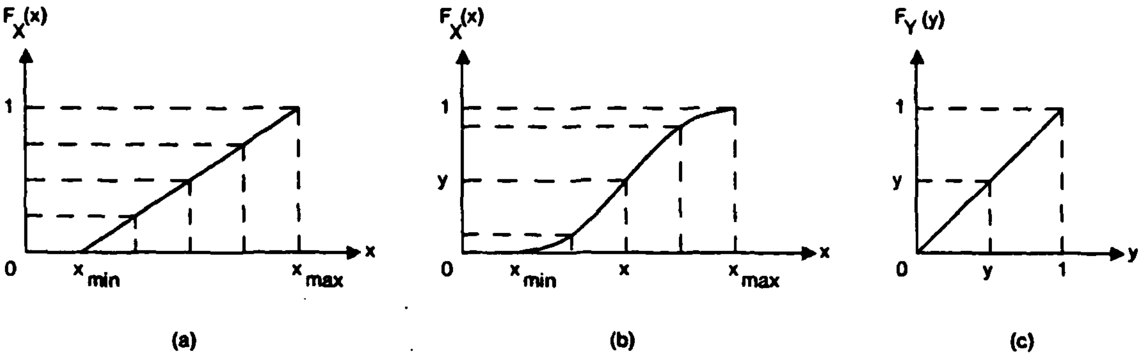


Figure 3-14: Examples of distribution functions.

illustrated in Figure 3-14 (b), the same method will produce partitions of different sizes. In the previous method, a hash or scrambling function is used with the hope that the hashed values have a distribution close to uniform. An ideal function is one which always maps the key values uniformly onto $[h_{min}, h_{max}]$. One such function, to be shown informally below, is the very same distribution function $F_X(x)$. (A formal presentation can be found in [92] which forms the main basis of this section. Related ideas, presented in the form of domain histogram, can also be found in [69].) Let the random variable X have the distribution function $F_X(x)$, as illustrated in Figure 3-14 (b). Let the random variable Y be equal to $F_X(X)$. For a fixed $y = F_X(x)$ in [0, 1], we have

$$\begin{aligned} F_Y(y) &= P[Y \le y] \\ &= P[F_X(X) \le y] \\ &= P[X \le x] \\ &= F_X(x) \\ &= y, \end{aligned}$$

and thus Y has uniform distribution between 0 and 1 (Figure 3-14 (c)).

Unlike the previous method, the transformation here is a nondecreasing function and therefore the ordering of keys is preserved. It is thus also suitable for external sorting where contiguous keys should be retrieved in the same partition. To handle cases in which distribution of keys is not known *a priori*, Weide [92] suggests its estimate by sampling the inputs to produce an empirical distribution function. His theoretic result is further confirmed by experimental runs on nonuniformly distributed data. The sample size used is constant and relatively small (between 10 and 30). Such experiments show that partitioning by the empirical distribution function can indeed be a practical method.

3.4.3 Discussion of the Hardware Requirements

We have shown two methods to obtain the appropriate partitions to be input to the primary memory for processing. Selection of the components of each partition can best be carried out by some kind of logic-enhanced secondary storage devices. Both Lin [69] and Shaw [83] suggest use of some kind of logic-per-track devices. With such hardware, a partition can essentially be brought to the memory in every disk rotation, assuming enough data can be retrieved. This would of course require a very fast handling of the retrieved data. As we shall see in Chapter 5, with a special-purpose hardware device of limited capacity, there are certain inherent I/O constraints which prohibit an arbitrary performance improvement. Therefore the amount of logic enhancement in the secondary storage devices should be enough to reach the ultimate goal of maintaining a continuous data flow.

3.5 Concluding Remarks

The new insertion and deletion procedures presented in this chapter offer two advantages: pipelining of sequences of inserts and deletes, and less storage and logic in each O-node. The latter advantage is probably more significant, since pipelining still would not be possible if the deletion assumption cannot be met. We have shown, however, that there are many concrete examples where deletion is performed only on inserted items. For such cases, the proposed scheme seems especially attractive.

With a large special-purpose device as assumed in this chapter, considerable performance improvements are obtained. The question is whether such a device is economically justifiable. In Chapter 5 we consider sorting and show that, even with a relatively small device, substantial speed-ups with respect to fast sequential sorting algorithms are possible. Since many important operations such as join and project can be reducible to sorting problems, an alternative to the design used in this chapter is to build a fast sorting device.

Chapter 4
Implementation Considerations

Some implementation considerations are contained in this chapter. We discusses the packaging of large tree structures on chips. The packaging method to be proposed requires only one type of fully utilized component chips. The wire lengths interconnecting component chips are shorter than previously known methods. The main result is based on a layout result for a new structure called a *linearized tree* that combines the characteristics of the linear array and the binary tree.

We also present a simple architecture and instruction set for the □-nodes. Recall □-nodes are the leaf nodes of the tree where data are stored and most computations take place. Signaling conventions in the O-nodes, as well as the exchange operations of the Δ-nodes, are discussed. Based on the simple design, pin requirements and some timing estimates are derived. A number of performance improvement results to be derived in later chapters are based on assumptions and estimates of this chapter. We note that we have used conservative estimates, preferring to underestimate rather than overestimate them.

4.1 Packaging Large Tree Structures

Since the appearance of the hexagonal systolic arrays for matrix computations [51], many other structures have been proposed to solve various kinds of problems. Such structures include the linear and orthogonal arrays, binary trees, etc. The simplicity and regularity of the interconnections in these structures are desirable properties for VLSI implementation. They ensure that communication will be local between neighbor cells and that the resulting layout will be compact. Such structures are modular in that their size can be extended as the chip capacity increases (with larger chip area or increased component density, or both). An interesting case to examine is when the structure needs more than one chip to be implemented. A large structure has to be decomposed into smaller components each of which can fit into one chip. The resulting chips are then linked together on one or more circuit boards. One immediate constraint in the partitioning of a given structure among chips is the pin limitation. Though chip capacity is increasing at a steady rate, the number of pins remains remarkably limited and is likely to improve only slowly. The nice properties we look for in a good partitioning strategy are very similar to those of systolic VLSI designs. A good VLSI design employs a *few* basic cells, interconnected together in a *simple* and *regular* way. Here we wish to use a *few* types of different *fully utilized* chips, for economic reasons. For the same reason, we would like to place as many chips on a circuit board as possible. In order to obtain a compact layout of component chips interconnected with short wires, the interconnection of chips should be *simple* and *regular*. This similarity shows that, if the layout of the component chips is one of the well-known structures proposed in earlier systolic algorithms, then packaging will likely be easy.

4.1.1 The Linearized Tree

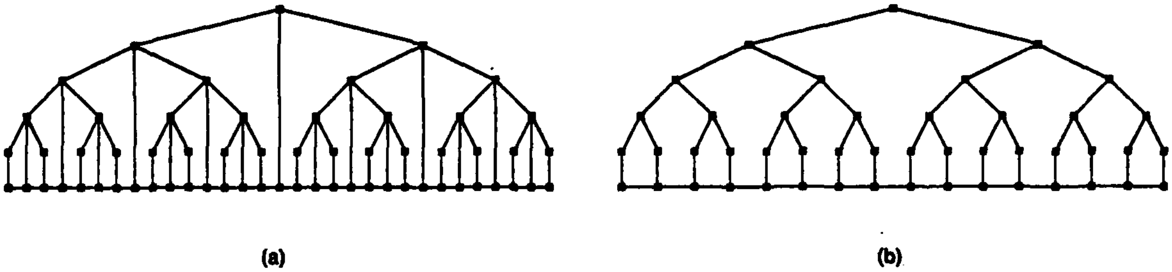
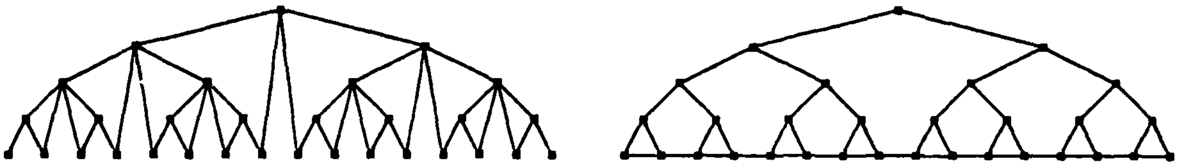


Figure 4-1: The linearized tree and a special case.

The *linearized tree* is a combination of a binary tree and a linear array, with each node in the binary tree connected to a node in the linear array in such a way that two nodes in the binary tree that are neighbors in the in-order traversal are connected to two neighbor nodes of the linear array. This is illustrated in Figure 4-1 (a). Figure 4-1 (b) shows a special case of a linearized tree where the linear array nodes are connected only to the leaves of the binary tree. In other words, only the leaves are linearized.



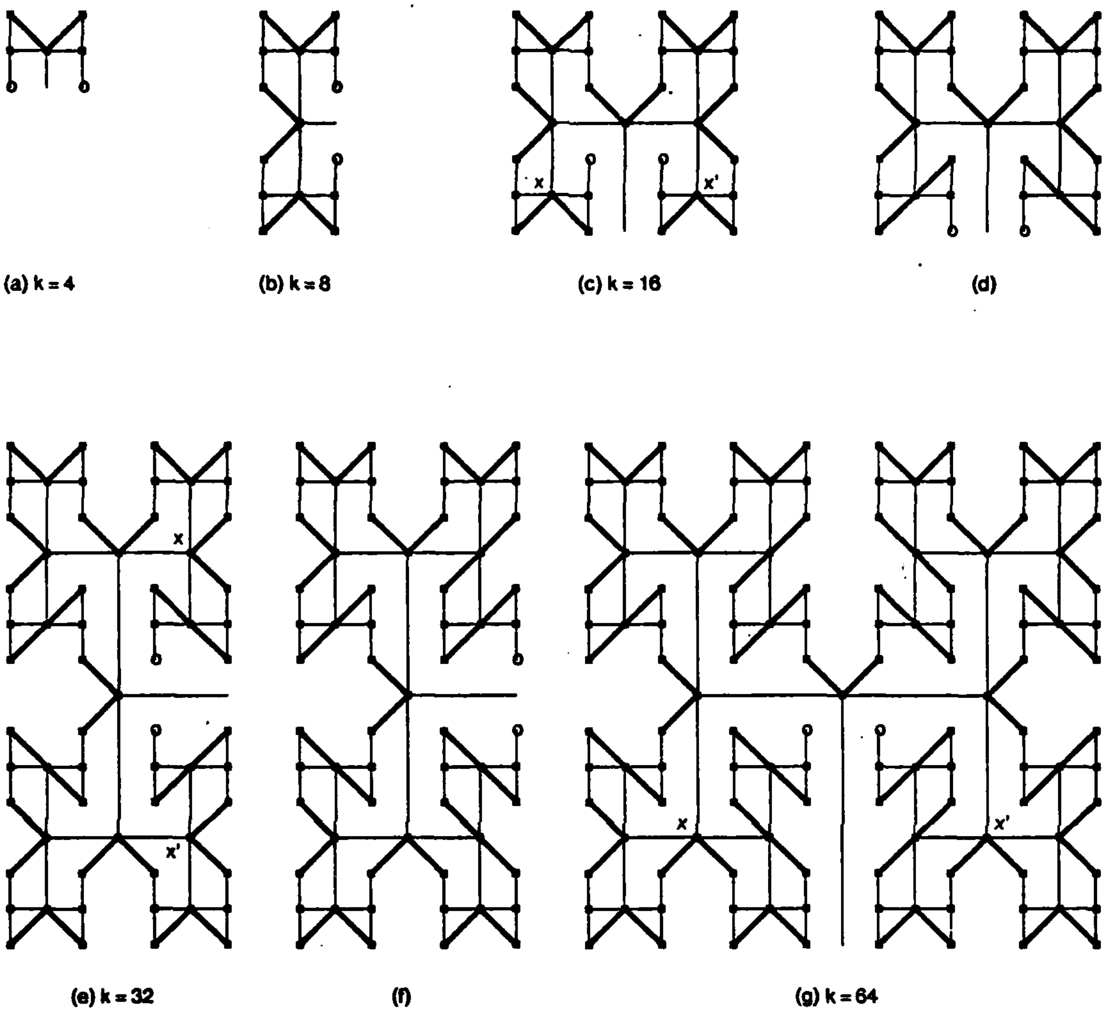
(a) (b)

Figure 4-2: The resulting structures after collapsing vertically connected nodes.

Let us now take the structures of Figure 4-1 (a) and (b) and collapse each pair of vertically connected nodes into a single node. The resulting structures are shown in Figure 4-2 (a) and (b). We will call the structure in Figure 4-2 (a) a *threaded tree*, borrowing the notation from Knuth [45]. The link connecting a leaf to an internal node will be referred to as a *thread link*. The threaded tree structure has also appeared in a work by Rosenberg, Wood, and Galil [78], under a different context (embedding of tree structures in trees), and by the name of "dree". Rem [77] discusses hierarchical structures and leaves open the question of whether to provide logic in the internal nodes of a binary tree or in its leaves. Bentley and Kung [9] for example, as in this work, decided to store data in the leaves where most of the computations take place. With the linearized tree represented in the form of a threaded tree, however, such decisions become unimportant, since the internal nodes and the leaves are connected through the thread links. The structure of Figure 4-2 (b) has been proposed for implementing priority queues [64], for maximal rate computation of recurrence relations [48], and for implementing a reduction language machine [72, 73].

4.1.1.1 Threaded Tree Layout

The threaded tree can be nicely laid out in the plane by applying a simple recursive procedure, as illustrated in Figure 4-3 (a) through (g). The threaded tree can be viewed as a binary tree plus the thread links. Its layout is similar to the classical H-layout for binary trees [75]. The recursive method to be presented applies rotation to part of the H-layout in such a way that all the thread links have constant length. Let k be a power of two denoting the number of leaves of a threaded tree. In Figure 4-3 (a) a threaded tree with k = 4 is shown. The thick lines represent the thread links, and the small circles denote the two end leaves which



(a) k = 4 (b) k = 8 (c) k = 16 (d)

(e) k = 32 (f) (g) k = 64

Figure 4-3: Laying out a threaded tree by a recursive procedure.

do not have thread links. Figure 4-3 (b), for k = 8, is obtained by combining (a) with its mirror image with respect to a horizontal axis and adding two more thread links. Similarly Figure 4-3 (c), for k = 16, is obtained by combining (b) with its mirror image with respect to a vertical axis and adding two more thread links. To obtain the layout for k = 32, we first obtain (d) by rotating part of (c) along xx' (to bring the two end leaves to the periphery) and then repeat the same construction as before. Similarly, for k = 64, we first obtain (f) from (e) by rotating part of (e) along xx' and then get (g). Thus, by construction, the resulting layout always has the property that all thread links are of constant length connecting neighbor nodes.

The layout of the threaded tree represents a self-contained result. It is interesting that this result can also be applied to the packaging of certain tree structures on chips. Such tree structures include the binary tree and the very same threaded tree.

4.1.1.2 Packaging a Large Binary Tree

Consider a large binary tree that needs a number of chips to be implemented. We repeat below the desirable characteristics of a packaging scheme.

1. Uses few types of component chips.
2. Fully utilizes each component chip.
3. Allows compact layout of the component chips.
4. Uses short wires to interconnect component chips.

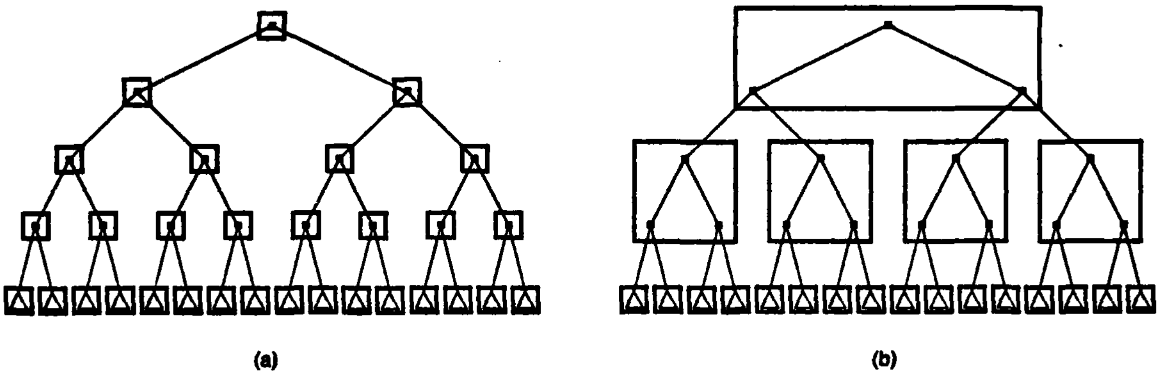


Figure 4-4: Two packaging schemes.

Two solutions are shown in Figure 4-4 (a) and (b). Each square at the bottom level, called a leaf chip, implements a sub binary tree (represented by the triangle). Solution (a) employs one chip, called an internal chip, for each internal binary tree node. The resulting component chip layout is still a binary tree, and can be compactly laid out on a circuit board. This solution, however, requires two types of component chips, and the internal chips are sub-utilized. Solution (b) attempts to put r internal binary nodes on an internal chip. The value of r, however, is restricted by pin limitation. The resulting chip layout is no longer a binary tree, but an (r + 1)-ary tree.

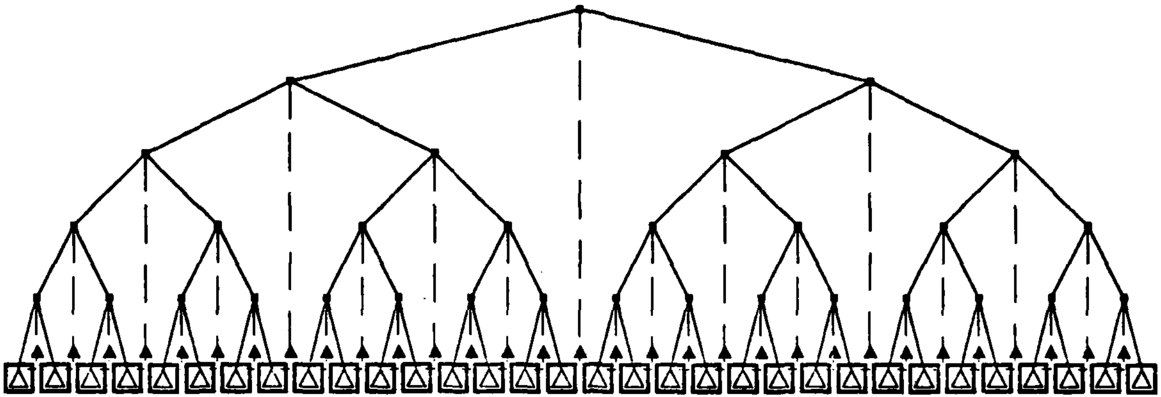


Figure 4-5: A new solution.

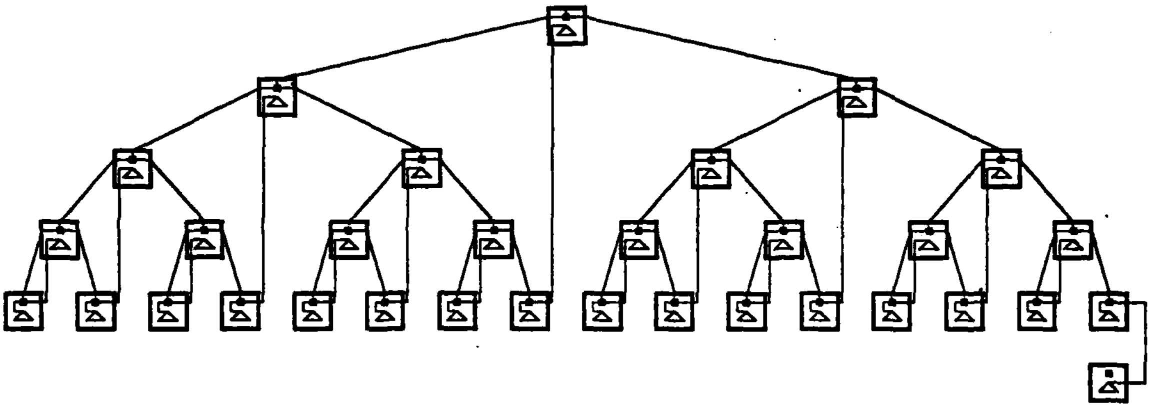


Figure 4-6: Chip layout after accommodating internal nodes.

For a binary tree of k leaf chips, we have exactly k-1 internal nodes that remain to be accommodated. Consider Figure 4-5. Let us make each leaf chip take one additional internal node as indicated by the arrow, that is, the in-order successor of the leaf chip considered as a single node. We obtain Figure 4-6. With this we have succeeded in employing only one type of chip which is fully utilized. Furthermore, the resulting layout is an incomplete threaded tree, or a left-threaded tree. We can thus apply the previous result and obtain Figure 4-7.

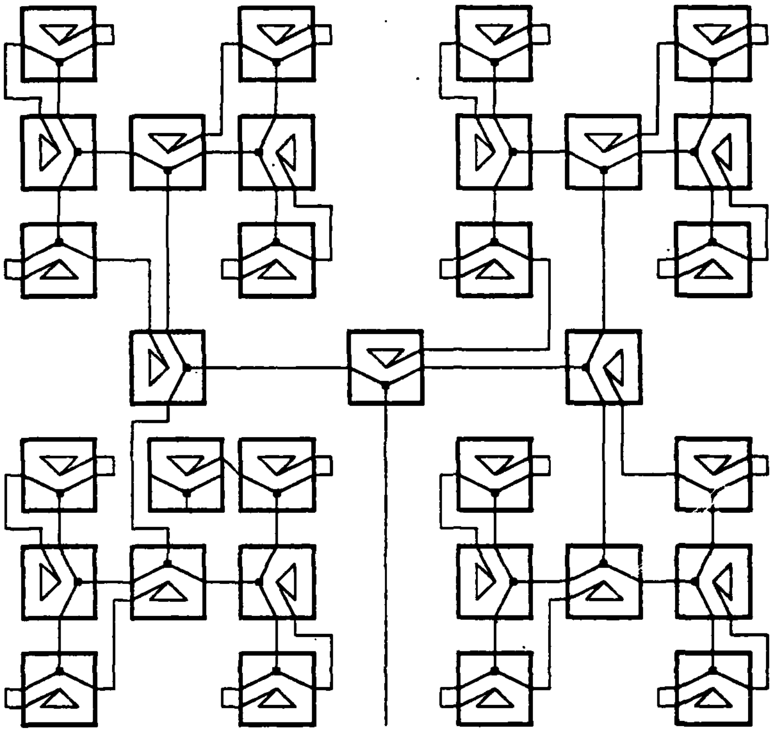


Figure 4-7: Final binary tree chip layout.

4.1.1.3 Discussion

The idea of incorporating an internal node to each leaf chip containing a sub binary tree is due to Leiserson [65]. Our method differs from his in the choice of the internal node to be combined into a leaf chip, and also in the application of rotation to appropriate parts during the layout process. While Leiserson's method essentially requires four times the interconnection wiring length of the classical H-layout, the proposed scheme uses only once. This is because the additional wires (not present in the H-layout) are all thread links which are of constant length connecting neighbor chips. Also, the proposed method is more general in that it can be extended to other kinds of tree structures as we shall shortly show.

The layout results are presented here in topological terms. In order to apply such results in practice, we should also take geometry into consideration. In a classical H-layout, as we proceed from the leaves to the root, the interconnection wires become longer. As a result, the drivers must be proportionally larger to drive them. The root of the tree, the largest node of all, would then communicate off the chip to the outside world without much performance penalty. In the proposed scheme, as well as in those using a single type of component chips, this is no longer true. If all the chips are identical then the nodes of the H-layout (of chips on circuit board) would be same in size, though all now communicate off the chip to other chips. It remains to be seen whether such a scheme would result in severe performance penalty.

4.1.1.4 Packaging a Large Threaded Tree

Figure 4-8 shows a large threaded tree that cannot fit into one single chip. Again, we make each leaf chip accommodate one additional internal node (its in-order successor). The resulting layout shown in Figure 4-9 is again a threaded tree, and hence we know how to lay it out. Figure 4-10 shows the final layout. As before, only one type of fully utilized chips is needed, and the thread links are all of constant length connecting neighbor chips.

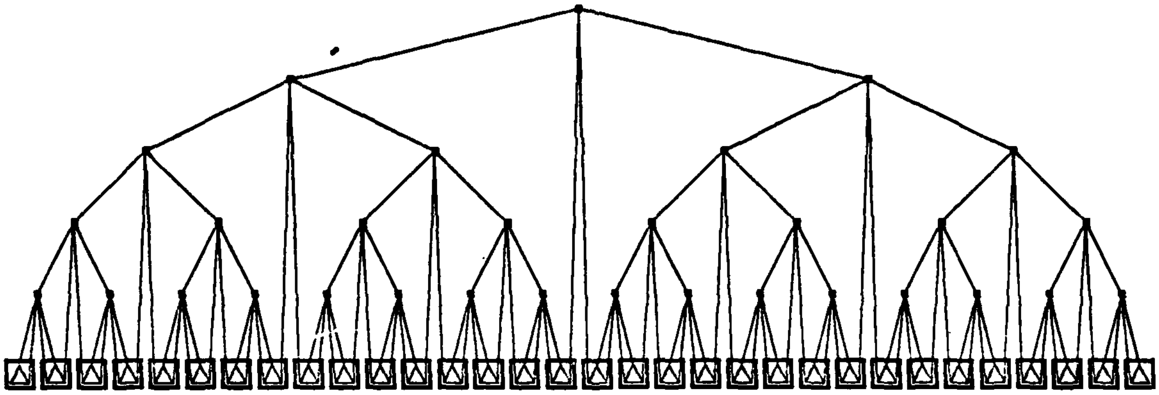


Figure 4-8: A large threaded tree.

4.2 Design Considerations

The main purpose of this section is to give some feeling for the complexity of an overall design, as well as to get some ideas on the design decisions involved. It will also provide us with rough timing estimates for the basic instructions and area estimates for implementing one □-node. We assume a datapath four bits wide for passing data between the nodes of the binary tree. Define a clock cycle as composed of two non-overlapping phases φ_1 and φ_2.

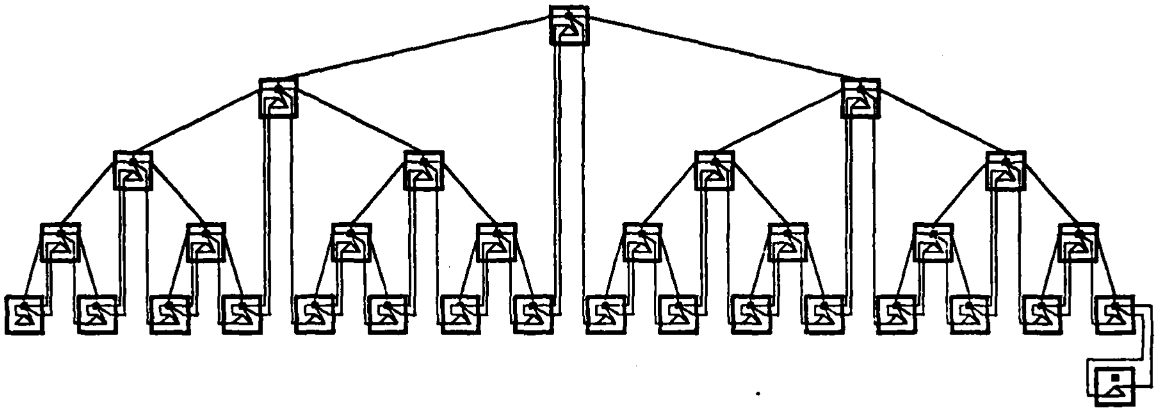


Figure 4-9: Chip layout after a.:commodating internal nodes.

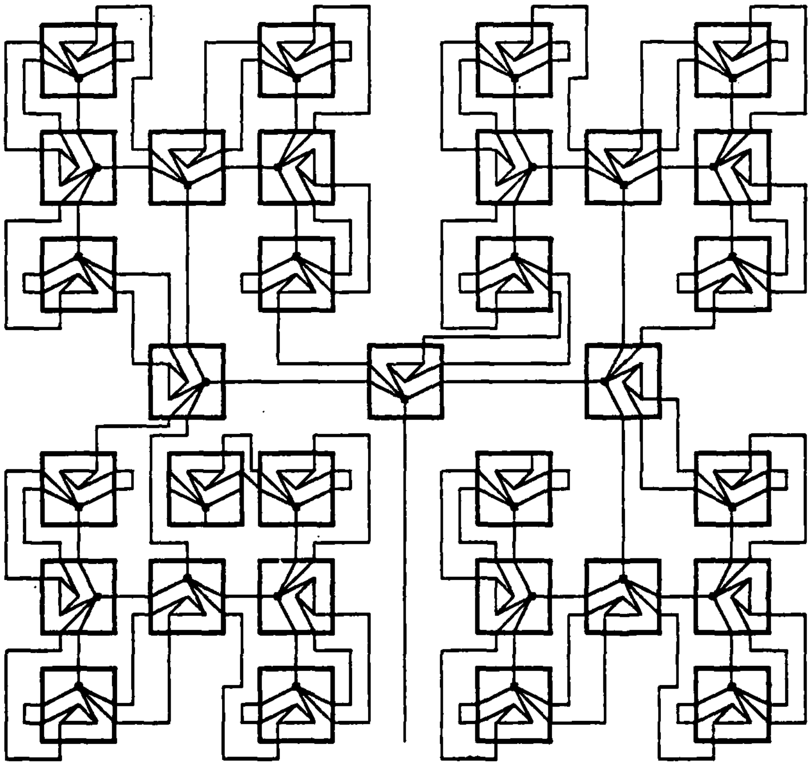


Figure 4-10: Final threaded tree chip layout.

4.2.1 Circle Nodes

Instructions and data, originated from the host or the interface controller, are inserted into the root ○-node and broadcast to the □-nodes. As. discussed earlier, some signaling conventions need to be devised. We use 2-cycle signaling where initiation and completion of operations are indicated by level transitions ìn two wires labeled Request and Acknowledge. The exact meaning of Request/Acknowledge depends of course on what is being requested and acknowledged. For example, Request can mean a request for the offspring nodes to take data, and Acknowledge sent back to a parent node can mean data have been taken. We shall use the following interpretation. A node sends Request to its parent node for data, and Acknowledge is sent to offspring nodes to indicate availability of data.

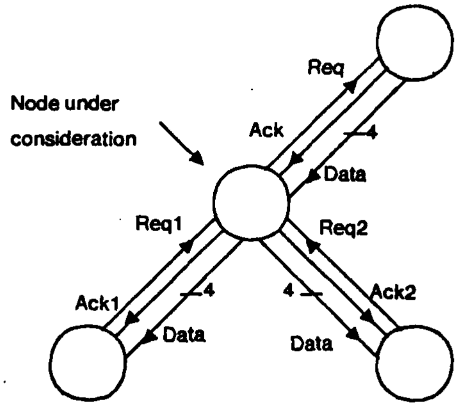


Figure 4-11: Data and signaling wires.

We assume each ○-node has a 4-bit storage latch. For expository purposes, Figure 4-11 gives the names of Request/Acknowledge wires for a particular node under consideration. The Resquest/Acknowledge signals transmit level transitions in either direction. When a node detects the occurrence of a level transition, say in Req_1 (from its left offspring), it may not be able to attend to the request immediately. This occurrence of level transition is recorded in a memory element called Requesting_1 to indicate a pending request from the left offspring. Similarly we have Requesting_2 for the right offspring. Requesting_1 and Requesting_2 are reset when the requests have been attended.

Present state	Inputs	Action	Next state
InitialState		Cause transition in Req	WaitForData
WaitForData	Transition in Ack	Latch data	HasData
HasData	Requesting1 and Requesting2	Cause transitions in Ack1 Ack2 Req Clear Requesting1 Clear Requesting2	WaitForData

Table 4-1: State transition table.

Consider the state transition table (Table 4-1). Initially all the O-nodes and □-nodes send requests to their parent nodes for data and go to the WaitForData state. A node may stay in the WaitForData state indefinitely, until an acknowledge signal is received from its parent. It then latches data and goes to the HasData state. Again, a node will remain in the HasData state until it has received requests for data from both offspring. Several actions are then taken. It sends acknowledge signals to both offspring and cancels the pending requests. It also sends a request to its parent for more data. Notice that when a node latches data after receiving an acknowledge signal, it can start sending the same data to its offspring nodes. Transmission of data thus has a head start over the transmission of the acknowledge signals eventually sent out when the right conditions are met. We assume that this delay can take care of the skew problem. Refering back to the state transition table, the system can be initialized with all the O-nodes and □-nodes in the WaitForData state and with Requesting_1 and Requesting_2 true. The interface controller, which acts as a parent node with respect to the root O-node, can thus start sending data and acknowledge signals to initiate the broadcasting. We assume that a 4-bit chunk of data can be transferred from a O-node to another in a clock cycle.

4.2.2 Square Nodes

Figure 4-12 illustrates the components of a □-node. Each □-node has eight 16-bit registers (Reg_0 through Reg_7), a ALU, eight flag bits ($Flag_0$ through $Flag_7$), and interfaces with the O-node and Δ-node. Reg_0 is the accumulator A which will participate in most of the instructions. Similarly, $Flag_0$ participates in most of the bit operations. The interface with the O-node consists of a 4-bit storage latch and Request/Acknowledge logic, just as any other O-node. A □-node interfaces with the Δ-node through two 16-bit registers, called LocOut and KeyOut. Data placed in these registers, normally consisting of a memory location and a key, are output through the tree of Δ-nodes.

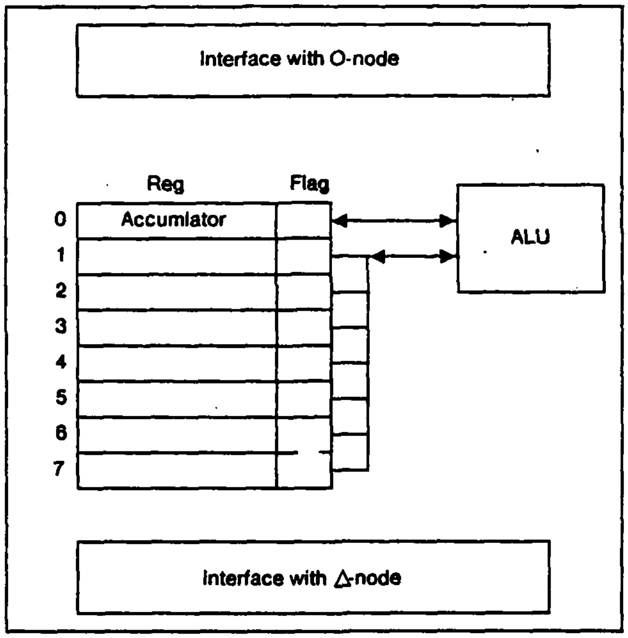


Figure 4-12: Components of a □-node.

An instruction has eight bits: a 5-bit op-code and 3-bit operand. Table 4-2 contains instructions that define contents of registers or flag bits, or move data to and from the accumulator. Table 4-3 contains the basic arithmetic and logical instructions.

Assembling an instruction requires two cycles. We assume that the instructions of Tables

Format	Description
Input i Content X	Define Reg_i with the input content X
InputCond i Content X	Same as above if $\mathrm{Flag}_0 = 1$
Load i	Set $A \leftarrow \mathrm{Reg}_i$
Store i	Set $\mathrm{Reg}_i \leftarrow A$
StoreLocOut	Set $\mathrm{LocOut} \leftarrow A$ if $\mathrm{Flag}_0 = 1$
StoreKeyOut	Set $\mathrm{KeyOut} \leftarrow A$ if $\mathrm{Flag}_0 = 1$
SetOne i	Set $\mathrm{Reg}_i \leftarrow 1$
SetZero i	Set $\mathrm{Reg}_i \leftarrow 0$
LoadFlag i	Set $\mathrm{Flag}_0 \leftarrow \mathrm{Flag}_i$
StoreFlag i	Set $\mathrm{Flag}_i \leftarrow \mathrm{Flag}_0$
SetFlag i	Set $\mathrm{Flag}_i \leftarrow 1$
ResetFlag i	Set $\mathrm{Flag}_i \leftarrow 0$

Table 4-2: Data movement instructions.

4-2 and 4-3, with exception of the input instructions (Input and InputCond), can be executed in two cycles. For these instructions, execution of the current one can overlap with assembling the next instruction. For the input instructions, however, the content to be input follow the current input instruction, and overlap is not possible. The total time for an input instruction is thus six cycles. A request for the next instruction will be sent when execution of the current instruction is complete.

Format	Description
Add i	$A \leftarrow A + Reg_i$
Subtract i	$A \leftarrow A - Reg_i$
ShiftUp i	$Reg_i \leftarrow A \cdot 2$
ShiftDown i	$Reg_i \leftarrow A / 2$
Compare i	$Flag_0 \leftarrow (A = Reg_i)$ $Flag_1 \leftarrow (A < Reg_i)$ $Flag_2 \leftarrow (A > Reg_i)$
CompareAnd i	$Flag_0 \leftarrow Flag_0 \wedge (A = Reg_i)$ $Flag_1 \leftarrow Flag_1 \wedge (A < Reg_i)$ $Flag_2 \leftarrow Flag_2 \wedge (A > Reg_i)$
CompareOr i	$Flag_0 \leftarrow Flag_0 \vee (A = Reg_i)$ $Flag_1 \leftarrow Flag_1 \vee (A < Reg_i)$ $Flag_2 \leftarrow Flag_2 \vee (A > Reg_i)$
And i	$Flag_0 \leftarrow Flag_0 \wedge Flag_i$
Or i	$Flag_0 \leftarrow Flag_0 \vee Flag_i$
Excl i	$Flag_0 \leftarrow Flag_0 \oplus Flag_i$
Not i	$Flag_0 \leftarrow \neg Flag_i$

Table 4-3: Arithmetic and logical instructions.

4.2.3 Triangle Nodes

Each Δ-node has storage for three pairs of keys and locations, corresponding to its own and those of its two offspring nodes. The function of a Δ-node is to rearrange these data such that it will retain the minimum key and the corresponding location. Details of this exchange operation will be discussed in Chapter 5. We assume that the exchange operation is the only operation performed by the Δ-nodes. Though it is intended to carry out the basic sorting step, it can also be used to output any two data items, loaded into the output registers LocOut and KeyOut.

4.2.4 Pin Requirements

The binary tree formed by the O-nodes is the mirror image of the binary tree of the Δ-nodes (Figure 4-13 (a)). By an "unmirroring" process, we can obtain the resulting binary tree structure of Figure 4-13 (b), which can then be packaged into chips by using the result of Section 4.1.1.2. A component chip of Figure 4-6 or 4-7 has the format shown in Figure 4-13 (c) where the two large overlapped triangles denote a subtree of Figure 4-13 (b).

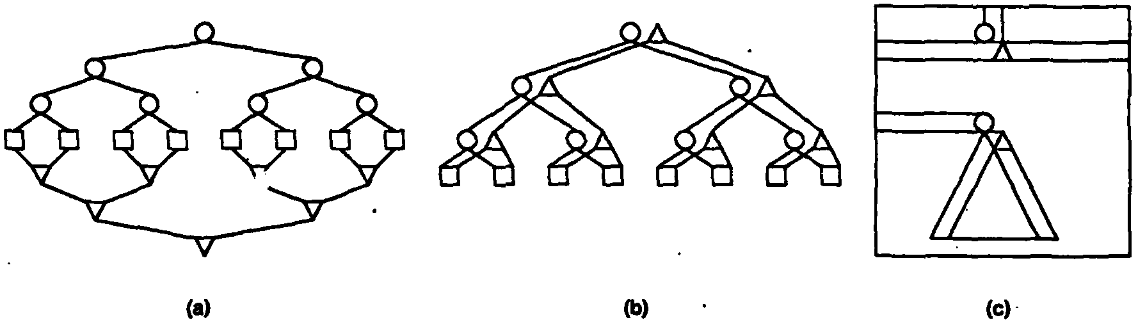


Figure 4-13: The unmirroring process and one component chip.

With a datapath of four bits wide for the data portion, we need at least

6 x (4 + 4) = 48 pins,

not counting those needed for power, ground, and clocks.

4.2.5 Timing and Area Estimates

Assuming a cycle of 100 ns, execution of most instructions, except the input instructions, takes 200 ns. An input instruction takes 600 ns. Four cycles, or 400 ns, are needed to pass a 16-bit quantity from a node to its neighbor. Some of these timings will be used in later chapters.

Consider the layout of a □-node as illustrated in Figure 4-14. We can get a fairly accurate estimate for the area of the eight 16-bit register array from the similar array in the design of OM2 [74]. This gives the approximate dimensions of 700 λ^2 by 440 λ for the register array.

[2] λ denotes the minimum feature size resolvable by the implementation process. A minimum-sized transistor has gate width of 2λ.

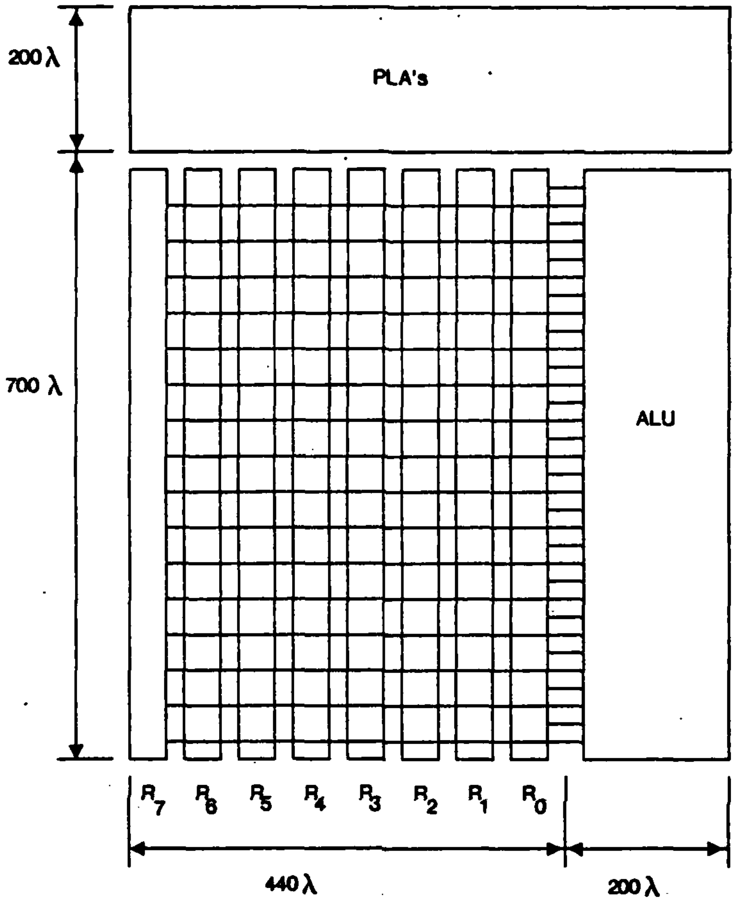


Figure 4-14: Layout of one □-node.

An adder cell used in [53] is about 50 λ by 100 λ. We estimate the area for the ALU to be 700 λ by 200 λ. The area for the PLAs used for control and decoding is assumed to be 200 λ by 440 λ. Using λ = 2.5 mm, the dimensions of a □-node is 2.3 mm by 1.6 mm. An internal node (in fact, a combination of a ○-node and a Δ-node) utilizes less storage than the □-node, and is assumed to require 2.3 mm by 1.2 mm. As shown in Figure 4-15, we can put eight □-nodes and eight internal nodes on a 10 mm by 6 mm chip. A more conservative estimate would put four □-nodes and four internal nodes on one chip.

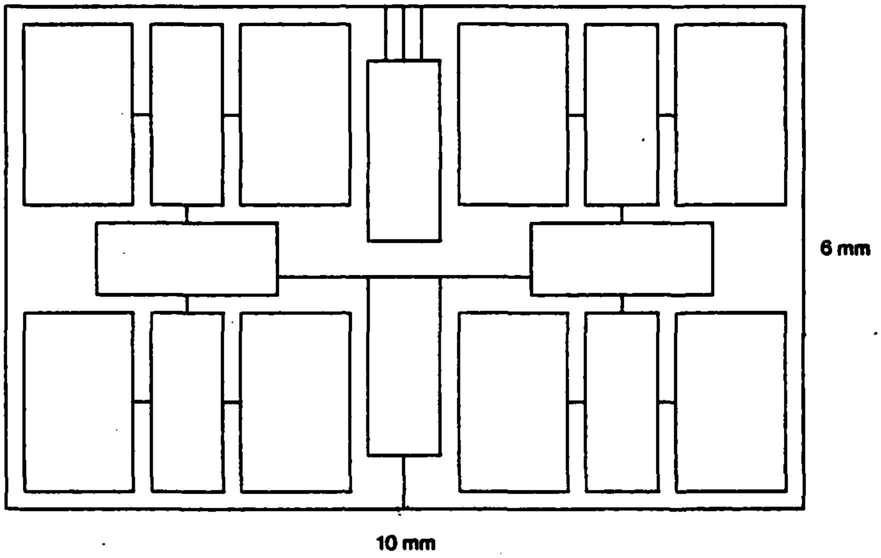


Figure 4-15: Layout of one component chip.

4.2.6 Some Examples

We now show how the instruction set can be used to solve database operations. We consider insertion, deletion, selection, and join. The sort operation is discussed in Chapter 5. In Section 6.6 the instuction set is used to solve a rectangle-intersection problem.

4.2.6.1 Insertion

The following instructions can be used to insert a content X into Reg_2 of a □-node, whose Reg_1 contains the FreePosition F (see Section 3.2.1.1).

Instruction	**Comment**
Input 0	Input position of free node
F	
Compare 1	See if selected
InputCond 2	Define Reg_2 of selected node
X	

4.2.6.2 Deletion

The following instructions delete the node with Reg_2 equal to X, and return it to the pool of free nodes. A FreePosition F will be loaded into Reg_1 of the deleted node.

Instruction	Comment
Input 0	Input X
X	
Compare 2	See if qualified for deletion
InputCond 1	Return to pool if deleted
F	

The deletion condition can be more complex. We merely replace the "Compare 2" instruction by a sequence of instructions expressing the condition, and set $Flag_0$ if the deletion condition is met.

4.2.6.3 Selection

To carry out the selection "retrieve all tuple ids (stored in Reg_3) where $Reg_1 = 1981$ and $10000 < Reg_2 < 20000$", we can use the following instructions.

Instruction	Comment
Input 0	Input constant 1981
1981	
Compare 1	
StoreFlag 1	$Flag_1 \leftarrow Reg_1 = 1981$
Input 0	Input constant 10000
10000	
CompareAnd 2	
LoadFlag 1	
StoreFlag 2	$Flag_2 \leftarrow Reg_1 = 1981 \wedge 10000 < Reg_2$
Input 0	Input constant 20000
20000	
CompareAnd 2	
LoadFlag 2	$Flag_0 \leftarrow Reg_1 = 1981 \wedge 10000 < Reg_2 < 20000$
Load 3	
StoreLocOut	Define output register with seleted tuple id

4.2.6.4 Join

Consider the join of relations A and B over some join attribute. Let Reg_2 and Reg_3 contain the join attribute (AttrA) and the corresponding memory location (LocA) of a tuple of A. The join operation can be carried out by the following sequence of operations for each tuple of relation B.

Instruction	Comment
Input 0	Input AttrB
AttrB	
Compare 2	See if AttrA = AttrB
Input 0	Input LocB
LocB	
StoreLocOut	Define output registers
Load 3	
StoreKeyOut	

4.3 Concluding Remarks

The packaging scheme presented in this chapter uses about one fourth of the total wire length with respect to the best previous packaging scheme. This result may be significant for packaging very large binary trees. To overcome the resistance and capacitance of long wires, large drivers are needed. Less interconnection wiring will thus reduce the number of large drivers. We note that the proposed scheme can be applied to the packaging of large linearized trees.

As an alternative to implementing the tree machine by custom design, we can also consider each node of the tree as implemented by a microprocessor. This approach is suggested in the X-tree project, in the context of a multiprocessor network design [80]. For the type of solution we are considering, the microprocessor as a tree node may be too powerful and general-purpose. The custom design approach has the advantage of requiring much less area for each node. To implement a tree machine of 64 leaves and 63 internal nodes, for example, we need only eight chips if our area estimate is correct. Even if we use the more conservative estimate, we still need a total of only sixteen chips. This represents a significant saving as compared to 127 microprocessor chips, each implementing one tree node.

A multiprocessor scheme for database applications proposed in [25] utilizes a number of microprocessors connected to a number of memories through a cross-bar switch. With such a scheme, the degree of parallelism may be limited. When the number of microprocessors and memories increases, the contention problem, among others, will degrade the performance. Also, in an economic sense, the number of switching elements may become rapidly infeasible. The custom design approach becomes more attractive when the chip capacity continues to increase. The modularity of the interconnection structure will allow an even larger degree of parallelism to be employed.

Chapter 5

Special-Purpose Hardware for Sorting

The problem of arranging a list of elements residing in main memory to form a sorted list is an important and basic problem. Sorting is frequently the dominant step in many algorithms. Knuth writes in the preface of [46] that he believes that virtually *every* important aspect of programming arises somewhere in the context of sorting and searching. He also notes the computer manufacturers' estimate that over 25 percent of the running time on their computers is spent on sorting, and that there are many installations in which sorting requires more than half of the computing time.

Sorting is also important in database applications. Once the required partition of data is brought into the main store, for example by some of the means as described in Section 3.4, operations such as project and join can all be treated as the sorting problem, with a subsequent linear match phase. The importance of the sorting operation in database applications is further strengthened by the findings of Blasgen and Eswaran that performing the join by sorting is among the two best of ten join methods [13].

Instead of running a classical sequential sorting algorithm on a conventional computer, we can use a special-purpose hardware device which accesses the memory directly to fetch the data to be sorted. Hardware-oriented sorting algorithms are described in [18] and [89], where quite impressive speed-up results are reported. By speed-up here we mean the ratio between the running times without and with the use of the special-purpose hardware. Such designs have concentrated mainly on the isolated sorting phase, without paying much attention to how data get to the sorting hardware. More significantly, no decomposition procedures are mentioned to handle the case where the number of data to be sorted exceeds the size of the special-purpose sorting device.

In this chapter we examine the problem of sorting a collection of numbers residing in the main memory with a special-purpose hardware device of a limited capacity. The important issues to be addressed, however, are not limited to the special case of sorting. In evaluating the performance of a high-speed special-purpose device, it does not suffice to consider the performance at an isolated, say chip, level. Since a special-purpose device is typically attached to a host, from which it gets the data to be processed and to which it outputs results, I/O considerations play an important role on the overall performance. The problem of I/O between the host and the special-purpose device becomes even more aggravated when a large computation cannot be handled all at once inside the device. Partitioning the computation into subcomputations implies the need to store and retrieve intermediate results. Often the amount of I/O needed will dictate the ultimate system performance, no matter how fast the special-purpose hardware can operate. Hong and Kung [37] have shown that the amount of I/O required for performing a Fast Fourier Transform of size n on a device of size s is $\Omega(n \log n/\log s)$. This implies that the maximum speed-up we can expect with respect to the $O(n \log n)$ sequential algorithm is of order log s.

We shall show that the I/O complexity to sort n numbers in the main memory by a special-purpose device of size s is $\Omega(n \log n/(t \log s))$, assuming the I/O bandwidth between the two sites is such that 2t numbers can be transmitted in one unit time. In Section 5.2 we present an algorithm for $t = 1$ which achieves this bound. These results are important in that they give us an idea on the order of magnitude of the speed-up we can expect, regardless of the details of any concrete design or implementation. Furthermore, their knowledge serves as a guideline as to what constitutes an optimal design. Use of I/O complexity to guide the design of special-purpose sorting hardware is illustrated in this chapter. First we obtain a lower bound result on the I/O complexity. Then we present an algorithm based on the multiway merge sort that achieves this bound. Two versions of the algorithm will be shown. The linear array version is based on the systolic priority queue [64], and the binary tree version is based on a hardware heap which can be implemented on the tree device described earlier. Both versions, to be referred to as systolic solutions, will be analyzed in detail. Then we discuss the design of the tree device and its interface with the host such that the overlap between computation and I/O is as nearly perfect as possible. Based on the analysis results and some implementation considerations discussed in Section 4.2, the expected performance improvements over two fast sequential sorting algorithms will be shown in Section 5.4.

5.1 I/O Complexity for Sorting

We now examine a lower bound result on the I/O complexity. First we define a model and some notation for its derivation.

5.1.1 Model Definition and Notation

We define the following model. We are given n numbers, assumed without loss of generality to be all distinct. We assume that these numbers are so small in physical size that, in order to see a number, we need to place a magnifying glass over it. We are given s magnifying glasses, with $s \leq n$. Moving a magnifying glass from one number to another is called a 1-move; t 1-moves constitute a *t-move*. Magnified numbers can be compared against one another and all previous comparison outcomes can be used to guide the selection of a next move. Comparison is the only operation we can perform on the numbers and it is assumed to be free. A t-move, however, costs one unit time. The goal is to produce a total ordering of the n numbers with a minimum amount of t-moves.

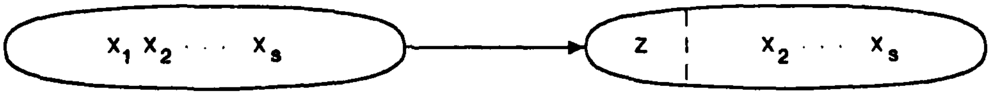


Figure 5-1: A 1-move replacing x_1 by z.

At any given time, at most s numbers can be magnified. A configuration of such a collection of numbers will be represented by an oval encircling the magnified numbers. A 1-move is illustrated in Figure 5-1, where the oval pointed to by the arrow indicates the new configuration produced by the 1-move. In this new configuration, the partition to the left of the dashed-line contains the newly magnified number.

5.1.2 A Lower Bound Result

We now show how information theoretic arguments similar to those in [46] can be used to derive lower bound results.

Theorem 1:

Let H be the minimum number of t-moves to sort n numbers, in the worst-case. We have
$H = \Omega(n \log n/(t \log s))$.

Proof:

Let the magnified numbers of a current configuration be $x_1, x_2, \ldots, x_s$. After any 1-move the newly magnified number, say z, can be compared against some or all of the other s - 1 numbers. At most s outcomes are possible, namely,

z is less than s - 1 numbers, and greater than none, or

z is less than s - 2 numbers, and greater than 1 number, or

z is less than s - 3 numbers, and greater than 2 numbers, or

.
.
.

z is less than none, and greater than s - 1 numbers.

Therefore, the number of outcomes after a t-move is s^t.

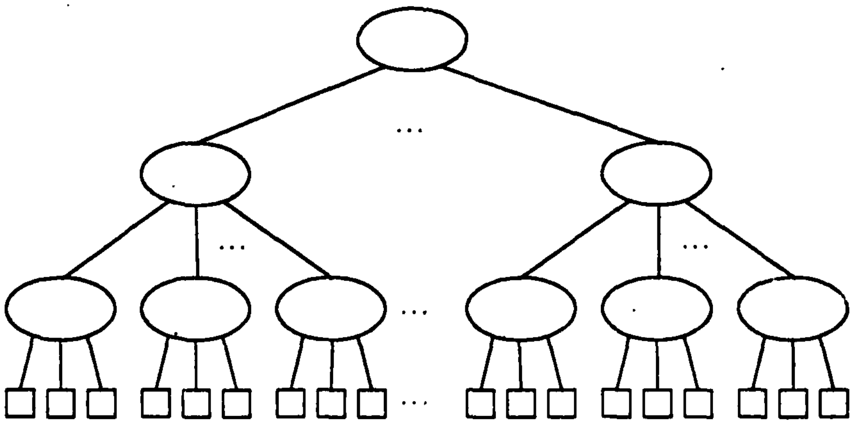


Figure 5-2: An s^t-ary t-move tree.

A sorting method under this model can be represented by an s^t*-ary t-move tree* as in Figure 5-2. Each internal node in this tree represents a t-move. The leaves of the tree contains a permutation of {1, 2, ..., n} indicating the ordering of the initial collection of n numbers to be sorted. The *t-move tree* is similar to the *comparison tree* described in [46], with the exception that here the height represents the number of t-moves instead of the number of comparisons made. Therefore we will not repeat the same kinds of arguments and give directly the following result. We are interested in finding the t-move tree that minimizes the maximum number of t-moves made (the best worst-case). As indicated in [46], this result turns out to be the same in the average case. This value, which we call H, is equal to the minimum height of the t-move tree. We have,

$n! \leq (s^t)^H$, or

$H \geq \lceil \log n! \,/\, (t \log s) \rceil$.

Using Stirling's approximation for $\lceil \log n! \rceil$, we have

$H \geq \lceil n \log n/(t \log s) + \text{lower order terms} \rceil$, and hence

$H = \Omega(n \log n/(t \log s))$.

□

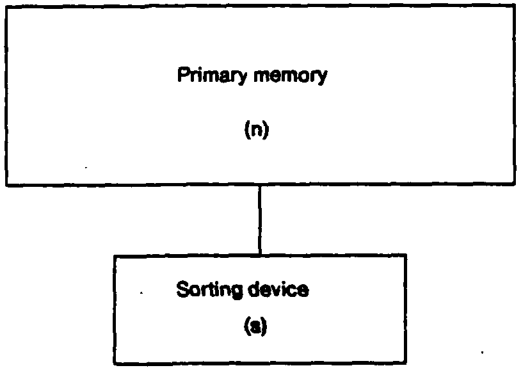


Figure 5-3: Interpretation of the model.

The model can be viewed as representing a sorting device of size s, attached to a memory of size at least n, with an I/O bandwidth between the two capable of storing t numbers and retrieving t numbers in a unit time (Figure 5-3). A more detailed interpretation of the parameter t will be given in Section 5.2.5.

It is interesting to note the difference between the results derived here and those by Hong and Kung [37]. They use a pebble game model to establish I/O complexity results for the fast Fourier transform, and several other problems such as matrix multiplication and odd-even transposition sort. Elements in the special-purpose device are represented by pebbles of a certain color. The number of pebbles characterizes the size of the device. An algorithm to carry out a specific computational task is defined by a computational graph. Given a set of rules that govern the placement and removal of pebbles of various colors on the graph, the goal is to arrive at a terminal configuration from an initial one, with a minimum number of application of certain rules. The initial and terminal configurations are coverings of certain nodes of the computational graph with pebbles of certain colors. Their lower bound results thus refer to a particular algorithm. We do not use a computational graph to characterize any particular algorithm. Our results, based on information theoretic arguments, are established for a class of algorithms. More specifically, the established lower bound results are for *any* sorting algorithm based on the decision tree model.

5.2 Upper Bound Results

The special case of $t = 1$ is of particular importance, and will be considered next. We present a sorting algorithm whose running time is of complexity $\Theta(n \log n/\log s)$. It is optimal since I/O alone has been shown to be $\Omega(n \log n/\log s)$.

5.2.1 Straight Multiway Merge Sort

Consider the problem of sorting n numbers stored in main memory, by using a special-purpose device of size s, with $s \ll n$. A straightforward solution is to use the sorting hardware only to produce $\lceil n/s \rceil$ sorted lists (called runs), and then use a sequential straight 2-way merge algorithm to combine the resulting runs. However, the speed-up may not be substantial for $n \gg s$. We wish to obtain a speed-up independent of n. First we examine a *sequential* s-way merge sort algorithm, where s input runs are combined into a single outpu run. The obvious way to merge s ascending runs is to examine the first element of each run and select the smallest element. This element is output and the process is repeated. This can be done with the aid of a heap of size s. The leading elements of s runs are used to form the initial heap. Then the minimum at the root of the heap is output and replaced by the successor element in its run. This process of replacing one value by another is called replacement selection. An infinity value can be appended to the end of each input run, so that merging will terminate gracefully (as suggested in [46]). In a sequential algorithm, after the smallest element of the heap has been output and replaced by its successor, the heap elements have to be rearranged and the next smallest element will be available after log s steps. In the systolic solutions to be described, however, this takes only a constant time. Thus we can intuitively expect a speed-up of O(log s). Let us first restate the decomposition procedure.

Problem:

Using a special-purpose device of size s ($s \ll n$), sort n numbers in memory to produce one ascending list.

Let q be an integer > 1 such that

$s^{q-1} < n \leq s^q$, or $\lceil n / s^q \rceil = 1$.

We have

$(q-1) \log s < \log n \leq q \log s$,

$\log n/\log s \leq q < (\log n/\log s) + 1$, and therefore

$q = \lceil \log n/\log s \rceil$.

Solution:

Define P(n,r) as the problem of merging r runs totalling n numbers into a single run. On a sorting device of size s, P(n,r) can be decomposed as first doing s-way merges on r runs to produce $\lceil r/s \rceil$ runs, and then solve P(n, $\lceil r/s \rceil$). Since P(n,1) requires no effort at all, the original problem P(n,n) terminates when only one single run is produced.

Value q has a special meaning. It represents the number of passes required by the s-way merge sort algorithm to put n numbers into a single sorted list. Each pass starting with r runs will end up with $\lceil r/s \rceil$ runs. Therefore the number of passes needed to merge n initial "runs" (each consisting of exactly one element) into a single run is

$$\lceil \log_s n \rceil = \lceil \log n/\log s \rceil = q.$$

5.2.2 The Linear Array Algorithm

A priority queue is a data structure where a collection of elements is maintained, with the possible operations of insertion, deletion, and minimum extraction. A hardware version of a systolic priority queue is described in [64]. It consists of a linear array of identical cells, each of which holds one element of the collection, with the minimum at one end of the linear array. The current smallest element will always be available in constant time, even though the remainder of the priority queue is still being reorganized.

Define a *basic cycle* of the systolic device as the time in which an extract-minimum/exchange step can be performed. We also assume that during a basic cycle two memory accesses can be made. Furthermore, we assume that during one half of a basic cycle data will flow into a cell of the systolic device, and during the other half data will flow out.

Let $T_s(n,r)$ be the time to do s-way merges on r runs totalling n numbers, and producing $\lceil r/s \rceil$ new runs. We assume that an infinity value is appended to the end of each initial run. $T_s(n,r)$ can be viewed as the time to introduce the first elements of each of the r runs to the systolic device, plus the time to perform a total of n extract-minimum/exchange steps. We have

$$T_s(n,r) = r + n \text{ basic cycles.}$$

To produce an initial run during the first pass, s numbers can be first inserted into the systolic device, and then the desired run is obtained by a sequence of extract-minimums. Thus $T_s(n,n)$ is simply

$$T_s(n,n) = 2\,n \text{ basic cycles.}$$

Let T(n,r) be the time to solve P(n,r) (as defined in Section 5.2.1), that is, the time to merge r runs totalling n numbers to form a single run. We have

$$T(n,r) = T_s(n,r) + T(n,\lceil r/s\rceil), \text{ and}$$

$$T(n,1) = 0.$$

Our problem is therefore to get T(n,n).

We will use the following lemma which can be proved by using techniques in [45].

Lemma:

If x is a real number and k a positive integer, then
$\lceil\lceil x\rceil/k\rceil = \lceil x/k\rceil$.

Theorem 2:

For the linear array algorithm, we have
$T(n,n) \leq (n + 1)\lceil\log n/\log s\rceil + n\, s/(s - 1)$ basic cycles.

Proof:

$$\begin{aligned}
T(n,n) &= T_s(n,n) + T(n,\lceil n/s\rceil)\\
&= T_s(n,n) + T_s(n,\lceil n/s\rceil) + T(n,\lceil\lceil n/s\rceil/s\rceil)\\
&= T_s(n,n) + T_s(n,\lceil n/s\rceil) + T(n,\lceil n/s^2\rceil)\\
&= T_s(n,n) + T_s(n,\lceil n/s\rceil) + T_s(n,\lceil n/s^2\rceil) + T(n,\lceil n/s^3\rceil)\\
&= T_s(n,n) + T_s(n,\lceil n/s\rceil) + T_s(n,\lceil n/s^2\rceil) + \ldots + T_s(n,\lceil n/s^{q-1}\rceil) + T(n,\lceil n/s^q\rceil),
\end{aligned}$$

where $q = \lceil\log n/\log s\rceil$, and we know $T(n,\lceil n/s^q\rceil) = T(n,1) = 0$.

Therefore, we have

$$\begin{aligned}
T(n,n) &= \textstyle\sum_{i=0}^{q-1} T_s(n,\lceil n/s^i \rceil) \\
&= \textstyle\sum_{i=0}^{q-1} (\lceil n/s^i \rceil + n) \\
&= n\,q + \textstyle\sum_{i=0}^{q-1} \lceil n/s^i \rceil \\
&\leq n\,q + \textstyle\sum_{i=0}^{q-1} (n/s^i + 1) \\
&= n\,q + n\,(1 + 1/s + 1/s^2 + \ldots + 1/s^{q-1}) + q \\
&= (n + 1)\,q + n\,(1 - 1/s^q)/(1 - 1/s) \\
&\leq (n + 1)\,q + n\,s/(s - 1) \\
&= (n + 1) \lceil \log n/\log s \rceil + n\,s/(s - 1) \text{ basic cycles.}
\end{aligned}$$

□

5.2.3 The Systolic Tree Device

The systolic tree device has been described in Chapter 3. Consider a tree device with s □-nodes. We saw in Section 3.3.3 that if each Δ-node of odd and even levels alternately executes the step: "Examine its own data and those of its two offspring, rearranging them if necessary such that it will contain the minimum", then the bottom half (i.e., the tree formed by the Δ-nodes) will become a heap-like structure after log(s) steps. Once the first elements of s runs are inside the hardware heap, we replace the minimum at the root Δ-node by the successor in its run, and this new element will be sifted up to its correct position. The next smallest element will be available at the root Δ-node in constant time, though the remainder of the heap is still being reorganized. A replacement selection step can thus be performed in constant time.

There are two ways to construct an initial heap of size k ($k \leq s$). One is to insert the k elements through the root ○-node until they reach the □-nodes, and then let the Δ-nodes produce the desired heap. Another way is to insert the k elements through the root Δ-node. In order to produce a balanced heap, the element to be inserted can be sent alternately to the left and right offspring (see [14]).

5.2.4 The Tree Algorithm

Define *basic cycle* in the same way as in Section 5.2.2. Furthermore, we assume that two tree levels can be traversed in a basic cycle.

Again, let $T_s(n,r)$ be the time to do s-way merges on r runs totalling n numbers, and producing $\lceil r/s \rceil$ new runs. Also, as before, assume that an infinity value is appended to the end of each initial run. We use the first method as mentioned above to form an inital heap of size s. That is, s elements are inserted through the root O-node until they reach the □-nodes, and then the Δ-nodes will produce the desired heap. $T_s(n,r)$ can be viewed as the time to produce $\lceil r/s \rceil$ initial heaps out of the first elements of each of the r runs, plus the time to perform a total of n extract-minimum/exchange steps. Therefore we have,

$$T_s(n,r) = r + \lceil r/s \rceil \log s + n \text{ basic cycles.}$$

$T_s(n,n)$ is a special case of $T_s(n,r)$ where we have n runs each with exactly one element. However, the infinity values which replace the minimums need not be read from memory, as in the general case. Instead this time can be used to input the s elements of the next "heap formation" step. In this way we can reduce the number of basic cycles roughly by half. Therefore,

$$T_s(n,r) = r + \lceil r/s \rceil \log s + n \text{ basic cycles, if } r \neq n \text{, and}$$

$$T_s(n,n) = n + \lceil n/s \rceil \log s \text{ basic cycles.}$$

Theorem 3:

For the tree algorithm, we have

$$T(n,n) \leq n(\lceil \log n/\log s \rceil + (1 + \log s)/(s-1)) + \lceil \log n/\log s \rceil (1 + \log s) \text{ basic cycles.}$$

Proof:

$$
\begin{aligned}
T(n,n) &= \sum_{i=0}^{q-1} T_s(n,\lceil n/s^i \rceil) \\
&= T_s(n,n) + \sum_{i=1}^{q-1} T_s(n,\lceil n/s^i \rceil) \\
&= n + \lceil n/s \rceil \log s + \sum_{i=1}^{q-1} (\lceil n/s^i \rceil + \lceil n/s^{i+1} \rceil \log s + n) \\
&= n + \lceil n/s \rceil \log s + (q-1)\, n + \sum_{i=1}^{q-1} \lceil n/s^i \rceil + \log s \sum_{i=2}^{q} \lceil n/s^i \rceil \\
&= n\, q + \sum_{i=1}^{q} \lceil n/s^i \rceil - 1 + \log s \sum_{i=1}^{q} \lceil n/s^i \rceil \\
&= n\, q + (1 + \log s) \sum_{i=1}^{q} \lceil n/s^i \rceil - 1 \text{ basic cycles.}
\end{aligned}
$$

Now

$$
\begin{aligned}
\sum_{i=1}^{q} \lceil n/s^i \rceil &\le \sum_{i=1}^{q} [(n/s^i) + 1] \\
&= q + n\,(1/s + 1/s^2 + \dots + 1/s^q) \\
&= q + n\,(1/s)\,(1 - 1/s^q) \,/\, (1 - 1/s) \\
&= q + (n - n/s^q) \,/\, (s - 1) \\
&\le q + (n - (\lceil n/s^q \rceil - 1)) \,/\, (s - 1) \\
&= q + n/(s - 1).
\end{aligned}
$$

Therefore, we finally have

$$
\begin{aligned}
T(n,n) &\le n\, q + (1 + \log s)\,(q + n/(s-1)) \\
&= n\, q + q\,(1 + \log s) + n\,(1 + \log s)/(s - 1) \\
&= n\,(q + (1 + \log s)/(2 - 1)) + q\,(1 + \log s) \\
&= n(\lceil \log n/\log s \rceil + (1 + \log s)/(s-1)) + \lceil \log n/\log s \rceil (1 + \log s) \text{ basic cycles.}
\end{aligned}
$$

□

The tree algorithm can be improved slightly to give yet another version, as follows. As mentioned earlier in Section 5.2.3, two methods can be used to produce an in:tial heap. One starts inserting the elements through the root O-node; the other introduces the elements directly through the root Δ-node, with the elements sent alternately to the left and right offspring to balance the heap. We can use the first method to produce initial heaps in the first pass and the latter one in subsequent passes. This will give ıne following expressions for $T_s(n,r)$.

$T_s(n,r) = r + n$ basic cycles, if $r \neq n$, and

$T_s(n,n) = n + \lceil n/s \rceil \log s$ basic cycles.

Therefore, we can obtain

$T(n,n) \leq (n + 1) \lceil \log n/\log s \rceil + n\, s/(s - 1)$ basic cycles.

In relation to the previous tree algorithm, the improvement is only roughly $n \log s/s^2$ basic cycles, which is not significant. The original tree algorithm has the advantage of allowing the handling of long sort keys by lexicographic sort, as describe in 3.3.3.1. The new tree version will thus be discarded.

5.2.5 A More General Interpretation of the Parameter t

We now give a more general way of interpreting the parameter t. The systolic device is designed in such a way that it will repeatedly carry out a sequence of computations, which we call a *basic computation cycle*. In the case of sorting, a basic computation cycle can be the insertion of a newly read number into the systolic device and the subsequent minimum extraction. The duration of a basic computation cycle will be referred to as the *basic computation time*. In carrying out a basic computation cycle, some quantities need to be retrieved from main memory and some others stored. We use the term *basic I/O cycle* to refer to the activity of performing these memory accesses including the address calculation by the interface controller. We use *basic I/O time* to denote the duration of a basic I/O cycle. Notice that if the communication path between the main memory and the special-purpose hardware allows p memory accesses to be done in parallel, then p basic I/O cycles can be performed during a basic I/O time. Similarly, if multiple systolic devices are used, then more than one basic computation cycle can be done in one basic computation time. t will then represent the number of basic I/O cycles that must be done during one basic computation time. That is,

t = p *basic computation time / basic I/O time*.

Notice that the basic I/O time depends on memory characteristics, as well as the interface controller design. We should first try to minimize basic I/O time. Then we try to design a systolic cell so as to achieve a unitary ratio between basic computation time and basic I/O time. (A more detailed discussion will be shown in Section 5.3.) A ratio less than unity means that the systolic device is needlessly fast. In case t > 1, we can employ t identical systolic devices so as to balance I/O and computation. In the case of sorting, we construct t systolic linear arrays (or trees) each of size s/t, and use t (s/t)-way merge sort algorithms. An example of t = 6 (with p = 3 and basic computation time / basic I/O time = 2) is illustrated in Figure 5-4.

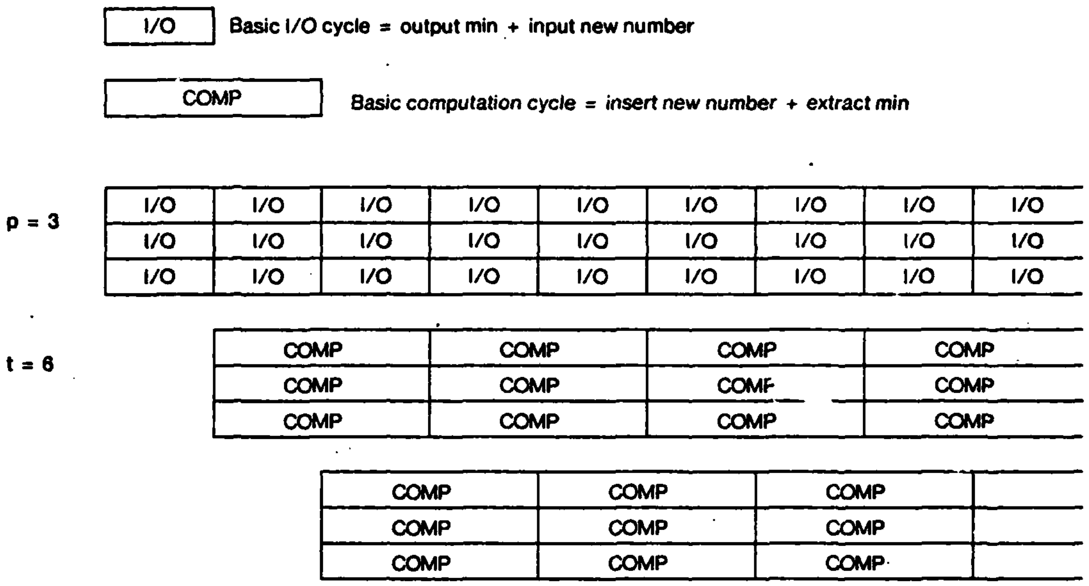

I/O
Basic I/O cycle = output min + input new number
COMP
Basic computation cycle = insert new number + extract min
p = 3
I/O
t = 6
COMP

Figure 5-4: An example of t = 6.

5.3 Considerations in the Design of the Interface Controller

A special-purpose device is typically attached to a host which provides the data to be processed and to which computed results are returned. An interface controller coordinates the data traffic between the host and the special-purpose device. As we have seen earlier, a *basic computation cycle* is a sequence of computations repeatedly carried out by the special-purpose device. The memory accesses needed in a *basic computation cycle* are carried out in a *basic I/O cycle*. There is a need to synchronize three closely related activities: the basic computation cycle, the basic I/O cycle, and the activity of the interface controller. For example, before a memory access can be initiated, the interface controller needs to compute its address which in turn may depend on the outcome of some step in a basic computation cycle. Similarly, a step in the basic computation cycle may have to wait for the conclusion of some data item being retrieved. A hypothetical situation is illustrated in Figure 5-5 (a). The ideal case is when no synchronization overheads are needed, as in Figure 5-5 (b).

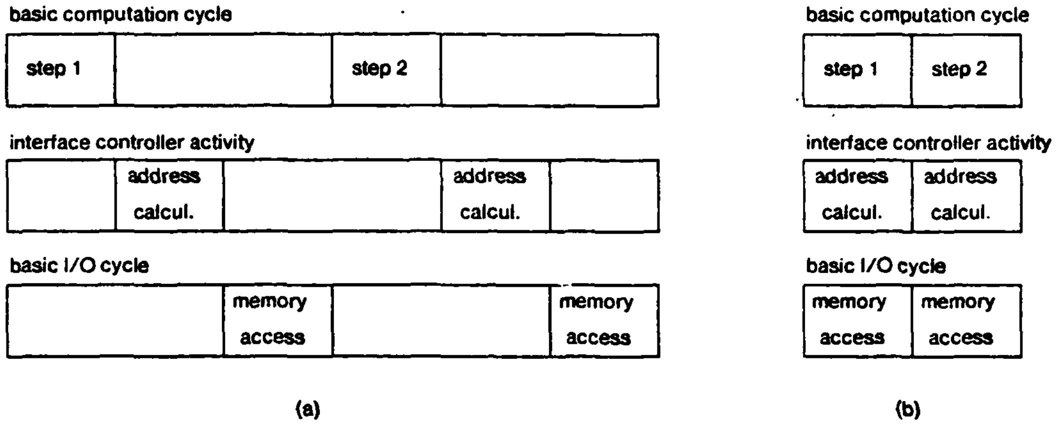


Figure 5-5: Synchronization overheads and an ideal situation.

Consider the sorting device described earlier. A basic computation cycle is the extraction of a current minimum of the elements in the device and the subsequent replacement by its successor element in the same run. Consider the tree algorithm where each heap is formed by entering the component elements through the root O-node. (The discussion that follows is easily adaptable to the linear array algorithm.) With each square node containing the data (Location, Key) where Key denotes the key on which sorting is being done and Location its memory location, Δ-nodes of odd and even levels alternately execute the following step: "Examine its own data and those of its two offspring, rearranging them if necessary such that it will contain the minimum key".

An earlier sorting example is repeated in Figure 5-6 (a) through (d). Though only the key values are shown, they are in fact accompanied by their corresponding locations. Assume that the datapath between Δ-nodes (4 bits) is much narrower than the length of a key or location (16 bits), and that some scheme of serial transmission is used. Let a unit of time be the time to transfer either a key or a location from a Δ-node to its offspring or parent node. The time to do comparisons will be assumed to overlap with the data transfer and will not be

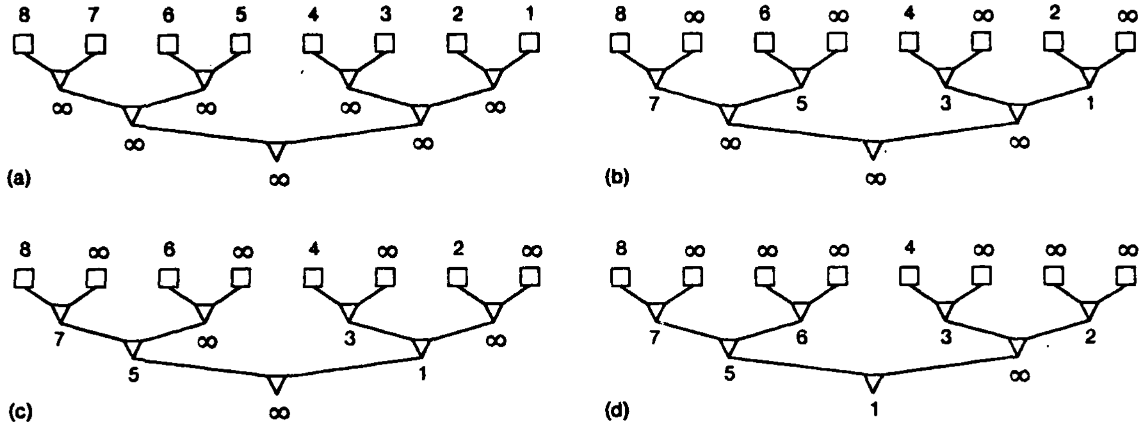


Figure 5-6: Groups of three nodes are compared against each other.

counted. To carry out the above mentioned step, four time units are required. During the first two time units, the keys and locations of two offspring nodes are moved to the parent node executing the step. In the two subsequent time units, the same information, possibly exchanged with those of the parent node, flow back to the respective offspring nodes. Since this step is to be carried out alternately by the odd-level and even-level nodes, a basic computation cycle would take eight time units in a straightforward design. Actually we can do better, since some of the activities can be overlapped.

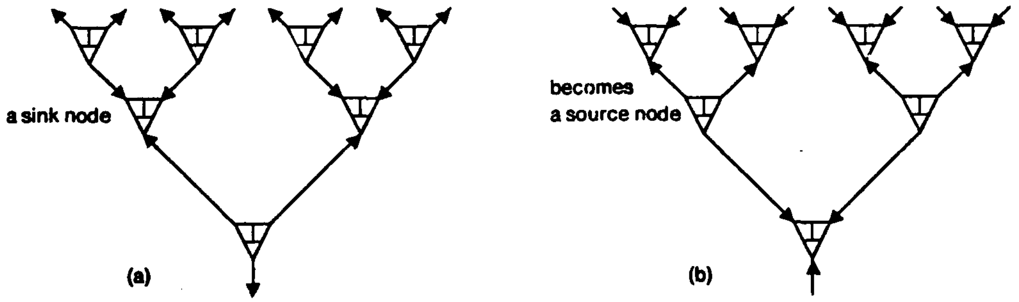


Figure 5-7: A sink node becomes a source node in the next time unit.

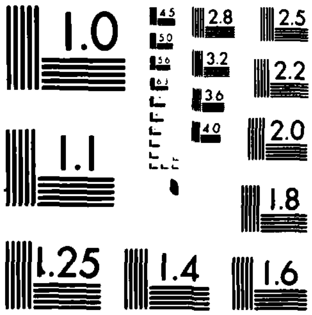
Each Δ-node will have storage for three pairs of keys and locations, corresponding to its own and those of its two offspring. Denote by *sink node* a node into which data (keys and locations) flow during two consecutive time units (see Figure 5-7 (a)). These data represent

AD-A112 542   CARNEGIE-MELLON UNIV. PITTSBURGH PA DEPT OF COMPUTER --ETC   F/G 9/2
ON A HIGH-PERFORMANCE VLSI SOLUTION TO DATABASE PROBLEMS.(U)
AUG 81   S W SONG   N00014-76-C-0370
UNCLASSIFIED   CMU-CS-81-142   NL

2 OF 2

END
DATE
FILMED
4-82
DTIC

11254



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963 A

the current updated data of the sink node and its two offspring. A sink node will compare key values and make appropriate local exchanges so that it will retain the minimum key and the corresponding location. The data of its offspring may be temporarily obsolete. A node acting as a sink node will become a *source node* during the two subsequent time units during which data flow out of it to the two offspring and to its parent node (see Figure 5-7 (b)). In this way it updates the data of the offspring and supplies a fresh copy of its own data to its parent. During two time units all the odd-level nodes function as source nodes and the even-level nodes as sink nodes. During the next two time units, the roles are reversed. A basic computation cycle thus takes four units of time.

5.3.1 Difficulty of Obtaining a Harmonious Flow

We now show a problem that may arise due to the need for synchronization. Figure 5-8 depicts the root Δ-node and its two offspring. Denote by Φ_1 the phase during which keys move and by Φ_2 the one in which locations move. The duration of each phase is a unit time. In the figure, only the key values are shown explicitly.

During one basic computation cycle the root Δ-node acts once as a source node and once as a sink node. The interface controller obtains the current minimum during phase Φ_1 (during which the root functions as a source node), and the corresponding location during phase Φ_2. With this location, it can calculate a new address (by adding some offset) and issue a memory read command to read in the successor key in the same run. These two activities need to be carried out serially and we thus see the difficulty of obtaining a harmonious flow. As soon as the root node outputs the key and location, shown respectively in Figure 5-8 (e) and (f), it will function as a source node and expects to input the next key value which is not yet available (Figure 5-8 (g)).

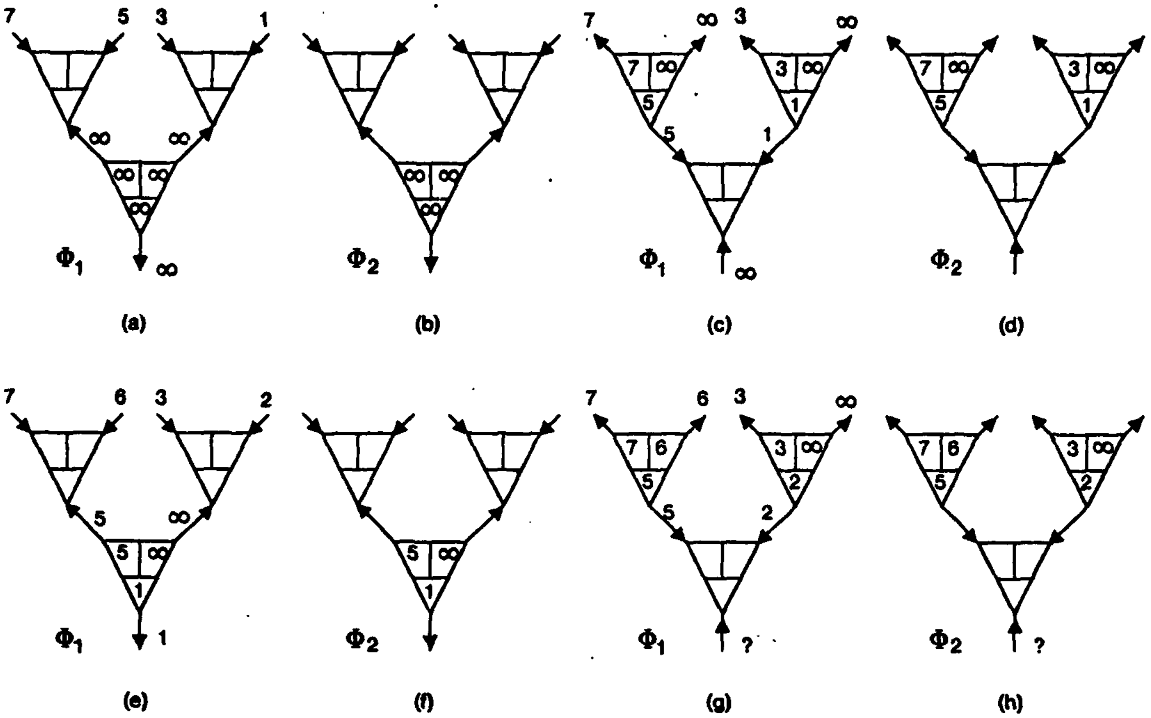


Figure 5-8: The difficulty of obtaining a harmonious flow.

5.3.2 A Simple Solution

One straightforward solution is to use buffering and prefetch all the s successor keys so that they become immediately available when needed. There is however a solution that does not need extra storage. Consider the root Δ-node in Figure 5-8 (c) and (d) corresponding to phases Φ_1 and Φ_2, respectively. At the end of phase Φ_2 both the minimum key and its location are available at the root Δ-node. Therefore if the interface controller were inside the root node instead of outside it, it could have started the input activities then, rather than two time units later. This observation suggests the following solution, illustrated in Figure 5-9 (a) through (h).

The interface controller has a storage for two buffers. The input buffer consists of input.key and input.location (which are initialized to ∞ and dummy location, respectively), and the output buffer consists of output.key and output.location. Let the basic computation cycle

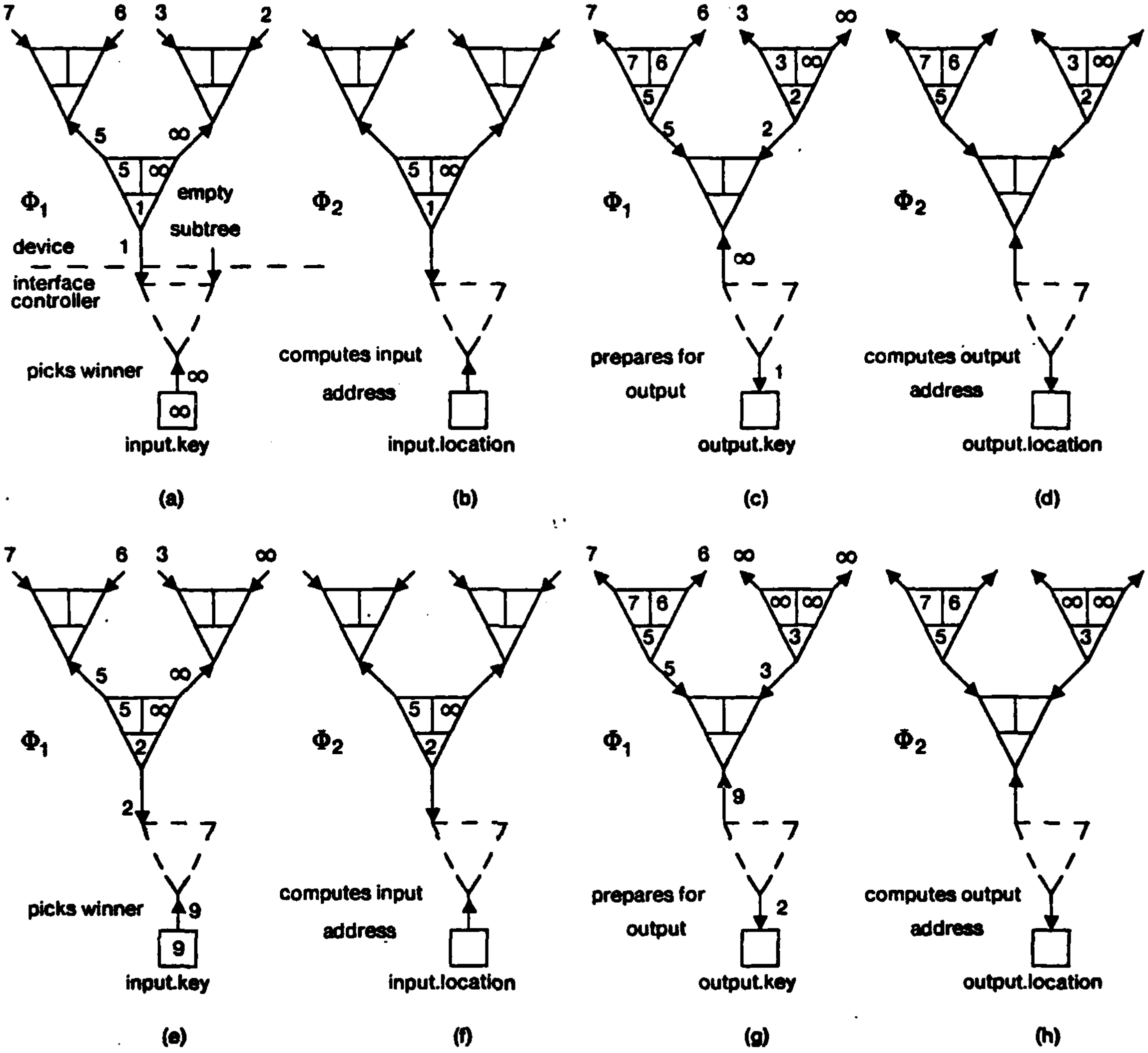


Figure 5-9: The interface controller acting as a new root node.

start with phase Φ_1 in which the root Δ-node acts as a source node (Figure 5-9 (a)). During Φ_1, the key value output from the device is compared against input.key and the smaller one located. Denote by *winner* the smaller key and by *loser* the larger one. In the subsequent phase Φ_2, the location of the winner becomes available a[illegible] be used to compute a new memory address to input the successor key of the winner. In the next half basic computation cycle the root Δ-node will act as a sink node. During phase Φ_1 the loser is fed back to the root Δ-node. The corresponding location is sent back in Φ_2. The actual memory read occurs during these same phases Φ_1 and Φ_2. The newly input key will be available in input.key by the end of phase Φ_2, ready to start all over again in the next basic computation cycle. What we have done is to obtain the second minimum key in the device while memory read is in progress and carry out a comparison outside the device. This can be best be viewed as treating the interface controller as a new root node whose right subtree is empty (or equivalently, contains always ∞ keys), as illustrated in Figure 5-9 (a). Figure 5-10 shows the timing diagrams of three activities. The ideal design is when the sum of Φ_1 and Φ_2 is equal to the memory access time.

	Φ_1	Φ_2	Φ_1	Φ_2
Basic computation cycle	Outputs key from device	Outputs corresponding location	Inputs key to device	Inputs corresponding location
Interface controller activity	Compares output key and stored.key	Computes address of winner's successor	Other activities	Computes output address
Basic I/O cycle	Write cycle to output winner of previous computation cycle		Read cycle to input next key into stored.key	

Figure 5-10: Timing diagrams showing overlapping of three activities,

5.4 Numerical Speed-Up Values

With the derived expressions for T(n,n), we can now get an idea of some numerical speed-up values. As we have just seen, the duration of a basic cycle is four time units or, using the estimates of Section 4.2.5 1.6 μs. Since two memory accesses are to be made in one basic cycle, this implies a memory cycle of 800 ns. We shall compute sorting times for various values of s, the size of the systolic sorting device. The sorting times will be compared against those of two fast sequential sorting algorithms, namely, Quicksort and Binsort. Quicksort is one of the best sequential sorting algorithms, as far as expected sorting time is concerned [34]. It is most highly regarded for practical applications. Binsort is a fast linear sorting algorithm, developed recently by Weide [92].

Consider the problem of sorting n numbers. Let

T_{tree} = Time taken by the tree algorithm (Section 5.2.4), and

T_{array} = Time taken by the linear array algorithm (Section 5.2.2).

We repeat the expressions for T_{tree} and T_{array} below.

$$T_{tree} = 1.6\,[n(\lceil \log n/\log s \rceil + (1 + \log s)/(s-1)) + \lceil \log n/\log s \rceil (1 + \log s)]\ \mu s, \text{ and}$$

$$T_{array} = 1.6\,[(n + 1) \lceil \log n/\log s \rceil + n\, s/(s - 1)]\ \mu s.$$

Let S_{tree} and S_{array} denote the respective speed-ups obtained in the tree and linear array algorithms, with respect to the particular sequential sorting algorithm in question.

5.4.1 Quicksort

Quicksort has an average-case running time proportional to n log n, though in the worst-case it is an $O(n^2)$ algorithm. We consider its average-case. Let

T_{quick} = Time Quicksort takes to sort n numbers.

On the DEC-KL10, the running time for Quicksort (from [92]) is

$$T_{quick} \cong 20\, n \log n\ \mu s.$$

We have Table 5-1, where we consider sizes $s = 2^6$ and 2^{12} (or 64 and 4096).

Several comments on this table seem in order. At a first glance, one might wonder, for example, how a 64-fold parallelism can give a speed-up of more than 64. This is possible because we are not speeding up the same sequential algorithm by a s-processor device, but rather a different algorithm is being used. In sequential sorting algorithms, because of overheads of control, book-keeping and data movement, the constant hidden in the big-O notation can be quite large. The power of the systolic solutions lies not only in the reduction of the sequential sorting time by a factor of log s, but also in the resulting small constant. This is so because of a much simpler control in the systolic solutions.

Both systolic algorithms have the same constant for the dominant term ($1.6\, n \lceil \log n/\log s \rceil$ μs). Therefore as n goes to infinity, both speed-up values S_{tree} and S_{array} approach

s	n	T_{quick}	T_{tree}	S_{tree}	T_{array}	S_{array}
64	$2^6 = 64$	7.7 ms	.12 ms	61	.21 ms	37
	$2^6 + 1$	7.7 ms	.24 ms	32	.31 ms	25
	$2^{12} = 4096$	983 ms	13.9 ms	71	19.8 ms	50
	$2^{12} + 1$	983 ms	20.4 ms	48	26.3 ms	37
	$2^{18} = 262144$	1 min 34 s	1.3 s	72	1.7 s	56
	$2^{18} + 1$	1 min 34 s	1.7 s	55	2.1 s	49
	$2^{24} = 16777216$	2 h 14 min	1 min 50 s	73	2 min 15 s	60
4096	$2^{12} = 4096$	983 ms	6.6 ms	149	13 ms	75
	$2^{12} + 1$	983 ms	13.2 ms	75	19.7 ms	50
	$2^{24} = 16777216$	2 h 14 min	53.8 s	150	1 min 21 s	100
	$2^{24} + 1$	2 h 14 min	1 min 21 s	100	1 min 47 s	75
	$2^{36} = 6.9 \times 10^{10}$	572 days	92 h	150	122 h	112

Table 5-1: Comparing systolic solutions with Quicksort.

$$20\ n \log n/(1.6\ n \lceil \log n/\log s \rceil) = 12.5 \log n/\lceil \log n/\log s \rceil\ \mu s.$$

giving the limits of 75 and 150, for s = 64 and 4096, respectively.

Recall that the number of passes to put n elements into one sorted list is given by $q = \lceil \log n/\log s \rceil$. For values of n in the ranges $[1, s]$, $[s+1, s^2]$, $[s^2 + 1, s^3]$, ..., the corresponding values for q are 1, 2, 3, ... The table is organized according to the number of passes needed. Thus, for example, for n in the range 65 to 4096, 2 passes are required. Note that the most friendly values of n are exact powers of s, i.e., log n/log s is an integer.

5.4.2 Binsort

Binsort [92] is linear algorithm, if the input data satisfy certain conditions, with the running time given by

$$T_{bin} \cong 180\ n\ \mu s.$$

Table 5-2 compares the systolic solutions with Binsort.

s	n	T_{bin}	T_{tree}	S_{tree}	T_{array}	S_{array}
64	$2^6 = 64$	11.5 ms	.12 ms	92	.21 ms	55
	$2^6 + 1$ $2^{12} = 4096$	11.5 ms 737 ms	.24 ms 13.9 ms	48 53	.31 ms 19.8 ms	37 37
	$2^{12} + 1$ $2^{18} = 262144$	737 ms 47.2 s	20.4 ms 1.3 s	36 36	26.3 ms 1.7 s	28 28
	$2^{18} + 1$ $2^{24} = 16777216$	47.2 s 50 min 20 s	1.7 s 1 min 50 s	27 27	2.1 s 2 min 15 s	22 22
4096	$2^{12} = 4096$	737 ms	6.6 ms	111	13 ms	56
	$2^{12} + 1$ $2^{24} = 16777216$	737 ms 50 min 20 s	13.2 ms 53.8 s	56 56	19.7 ms 1 min 21 s	37 37
	$2^{24} + 1$ $2^{36} = 6.9 \times 10^{10}$	50 min 20 s 143 days	1 min 21 s 92 h	37 37	1 min 47 s 122 h	28 28

Table 5-2: Comparing systolic solutions with Binsort.

Observe that even with a size of s = 64, the systolic solutions can still beat Binsort.

Because of complex book-keeping and control overheads, the constant of proportionality that expresses the running time of Binsort is quite big. Since Binsort is linear, and the systolic solutions are O(n log n/log s) algorithms, Binsort will win for sufficiently large values of n, with S_{tree} and S_{array} approaching

$$180\, n/(1.6\, n \lceil \log n/\log s \rceil) = 112.5/\lceil \log n/\log s \rceil.$$

Notice the successively decreasing values of S_{tree} and S_{array} while n increases.

It is interesting to find the break-even point when Binsort will start to beat the systolic solutions. For s = 64 and log s = 6, this will happen when

$$1.6\, n \log n/\log s > 180\, n, \text{ or}$$

$$n > 2^{675} \cong 10^{202}.$$

It is also interesting to obtain the largest value of n that still allows, say, a 10-fold speed-up. For this purpose, we make

$$16\, n \log n/\log s = 180\, n, \text{ or}$$

$$n = 2^{67} \cong 10^{20}.$$

Thus the systolic solutions can give substantial improvements for all practical ranges of the number of elements to be sorted.

5.5 Concluding Remarks

We have presented and analyzed the linear array algorithm and the binary tree algorithm. A comparison of the linear array and tree algorithms seems in order. The dominant term in the sorting times is n log n/log s basic cycles in both cases. The difference in lower order terms in the linear array algorithm and the tree algorithm is approximately (1 - log s/s) n basic cycles. This difference arises because the tree-structured device possesses both an input tree and an output tree, thereby allowing overlapping of I/O during the first pass. The tree algorithm performs better in one pass out of a total of $q = \lceil \log n/\log s \rceil$ passes. Thus a significant improvement is possible if q is very small, say one or two passes. As shown in Tables 5-1 and 5-2, for a size s = 4096, only one or two passes are needed to cover all practical ranges of n (say, up to a few million). For a device of such a size, the tree algorithm may be the choice over the linear array.

However, one important finding in this chapter is that, even with a size of only 64, the speed-ups are already substantial over the sequential algorithms. We may thus choose to build a sorting device of such a size, which is attractive for economic reasons. For small values of s, the performance difference between the two systolic solutions is not significant in a large portion of the practical range of n. Because of the simpler structure of the linear array, it should probably be preferred.

Chapter 6

Applications

We have examined a special-purpose device capable of supporting the relational join, search, sort, and some other basic operations. It can also maintain a collection of items with the possible operations of insertion and deletion. In the following we examine some problems that arise in design databases and some special databases such as chemical and geographic databases. We show that solutions to such problems depend heavily on a small set of basic operations.

First we examine two problems of detection of three-dimensional patterns in a large structure. In the first problem the pattern and structure are composed of a number of points. In the second one they are composed of line segments. We show that both problems can be solved by using essentially sorting and the relational join. Three other problems to be considered are detection of pairwise intersections of rectangles, a multidimensional search problem, and the containment problem. In addition to sorting, the solution to the first problem uses a data structure that maintains a dynamic collection of objects, with the operations of insertion, deletion, and search. Search is also the main ingredient in the solution of the latter two problems. While it is a widely known fact that sorting is useful in many applications, we emphasize that it is being used in some unexpected manner as some of the following examples show.

6.1 Detection of Three-Dimensional Patterns of Points

The problem of detecting three-dimensional patterns of a number of points, defined by their Cartesian coordinates and a specified set of attribute values, in a similarly defined larger structure is an important problem in many special databases. We will consider this problem in the context of detection of all occurrences of a certain pattern of atoms in a large molecular structure, an important database problem with applications in pharmacology, X-ray crystallography, and chemical documentation [67].

6.1.1 Statement of the Problem

A molecular structure of n atoms is given by a list of atomic coordinates $s_i = (x_i, y_i, z_i)$, i = 0, 1, ..., n-1. Each atom possesses a number of attributes such as atomic number, number and types of bonds, etc. For each of the n atoms in the structure we are also given a set of attributes. For simplicity we assume these to be represented by a single real number S_i, i = 0, 1, ..., n-1. Similarly a pattern of m atoms is defined by the atomic coordinates $p_j = (x_j, y_j, z_j)$ and their corresponding attribute values P_j, j = 0, 1, ..., m-1.

The problem is to determine all subsets of size m of the n atoms of the structure such that these match the m atoms in the pattern, to within some prefixed threshold values. More precisely, given non-negative threshold values ε_1 and ε_2, we want to determine all possible sets of indices k[0], k[1], ..., k[m-1] such that for j = 0, 1, ..., m-1,

(i) $|S_{k[j]} - P_j| \leq \varepsilon_1$ and

(ii) $|s'_{k[j]} - p_j| \leq \varepsilon_2$,

where $s'_{k[j]} = T(s_{k[j]})$, with T representing any Euclidean transformation consisting of rotation, translation, and mirror image, or a finite composition of these.

The first condition indicates that the attributes of the m selected atoms match those of the pattern to within threshold ε_1. The second condition indicates that, after some transformations, the positions of the m selected atoms of the structure coincide with those of the pattern to within threshold ε_2.

An extension to this problem is to include also scale in the set of allowable Euclidean transformations. This will be discussed in Section 6.2 in the context of three-dimensional shape matching.

6.1.2 Description of a Previous Work

The above problem has been considered in [67]. An algorithm has been proposed to report all occurrences of the pattern in the structure. If no complete matches are found, it reports the maximal subset for which a match exists. The algorithm first selects those atoms in the structure which are eligible to match some atom in the pattern. Once the candidate set of atoms is obtained, all its possible subsets of size m are then tested for congruence with the pattern to within the specified threshold values. If all the n atoms in the structure are found to be eligible, then the number of tests to be done will be $\binom{n}{m}$. The CPU time for this program to detect matches of a 5-atom pattern in a structure of 100 atoms was reported to be more than two hours. It is the latter phase of the algorithm, of generation and testing for congruence of all possible combinations, that we are going to improve.

6.1.3 Summary of New Complexity Results and Assumptions

The time complexity of the matching algorithms we shall describe depends on the distribution of points in the pattern and in the structure. We present two improved sequential algorithms whose worst-case time complexities are $O(n^2 \log m)$ and $O(n^2)$, for any distribution of points in the pattern and in the structure, given that assumptions 1 through 3 below are satisfied. The running times of both algorithms are dictated mainly by the computation of many join operations. In the second algorithm, the time complexity becomes $\min\{O(n^2), O(n\ m^3)\}$, if the points in the pattern obey certain conditions to be defined later. For a uniform distribution of pattern points, the time complexity will be at most $O(m\ n)$. This algorithm requires an $O(n^2 \log n)$ preprocessing of the structure which consists essentially of the sorting of n sequences of n elements each. These algorithms are therefore nice candidates to be solved by some special-purpose hardware capable of performing the join and sorting operations.

Let us now present a general framework that not only applies to this particular problem, but also to other types of matching problems such as the 3-D shape matching problem to be discussed later. The terms atom and point are used interchangeably. We use P and S to denote the pattern and the structure, respectively. We make the following general assumptions.

G1. There exists a point j_0 in the pattern P that, together with two of its "neighbors" j_1 and j_2 and some other point j_3, determine all the remaining m−1 points of P.

G2. We define "neighbor" in such a way that the number of "neighbors" of each point of P or S is bounded by a constant independent of m or n.

G3. The position vectors s_i, $i = 0, 1, \ldots, n-1$ are all distinct and the same should hold with respect to p_j, $j = 0, 1, \ldots, m-1$. In other words, no two points in S (and also in P) occupy the same position.

G4. We are interested only in detecting subsets of size m of S which match all the points of P. If no complete matches are found, then we will not report the maximal subset of the pattern for which a match exists.

The general idea is to try each of the n points of S as matching the point j_0 of P. Having fixed one such point of S, the number of choices of the three other points of S that match j_1, j_2, and j_3 will be bounded by a constant. Each matching process will consist of determining which points of S match the remaining m−3 points of P. If each matching process can be carried out in O(n) time, the whole matching problem will then be solved in $O(n^2)$ time.

For the present problem, we define "neighbor" as the nearest neighbor using the Euclidean-distance metric. Assumptions G1 through G4 can be re-stated as follows.

1. There exists an atom j_0 in the pattern, whose two nearest neighbors, j_1 and j_2, and some other atom j_3 are such that the vectors

   $p_{j_1} - p_{j_0}$, $p_{j_2} - p_{j_0}$, and $p_{j_3} - p_{j_0}$

   are linearly independent. (In the two-dimensional case, we need an atom j_0 whose nearest neighbor j_1 and some other atom j_2 are such that $p_{j_1} - p_{j_0}$ and $p_{j_2} - p_{j_0}$ are linearly independent.)

2. The number of nearest neighbors of each point in P or S is bounded by a constant.

3. Same as G3.

4. Same as G4.

Consider the distances between each point of P or S and its nearest neighbor. Let d_{max} and d_{min} denote the maximum and minimum of such distances. A sufficient condition for assumption 2 to hold in the 2-D case is

$$d_{max} \leq c\, d_{min}$$

for some small constant c independent of m or n. This is easily proven as follows. Let #NN denote the number of nearest neighbors of a point A under consideration. In the worst case, all the nearest neighbors of A are spaced by d_{min} on a circumference at distance d_{max} from A. Then

$\sum$ cords $\leq$ perimeter of the circumference, or

$\# NN\, d_{min} \leq 2\pi\, d_{max}$, and therefore

$\# NN \leq 2\pi\, d_{max}/d_{min} \leq 2c\pi$.

For the sake of simplicity in the presentation, we assume all the attribute values of the atoms of the structure and of the pattern to be equal, thereby eliminating condition (i) for the match. In a real situation, this condition of course should be taken into consideration as it will cut down the number of candidate atoms for the match. Without loss of generality, we assume the threshold value ε_2 to be equal to zero. We use s_{ij} and p_{ij} to denote the distance between points i and j in the structure and pattern, respectively. (Recall that the bold-faced $\mathbf{s}_i$ and $\mathbf{p}_j$, with a single subscript, are used to represent position vectors of atoms in the structure and pattern, respectively.)

6.1.4 A Worst-Case O(n^2 log m) Algorithm

For ease of presentation and illustration, we will describe the algorithms for the two-dimensional case. The complexity results to be derived are valid in the three-dimensional case.

Consider a pattern of m points 0, 1, ..., m-1. Without loss of generality let 0 be a point whose nearest neighbor 1 and some other point 2 are such that the vectors $\mathbf{p}_1 - \mathbf{p}_0$ and $\mathbf{p}_2 - \mathbf{p}_0$ are linearly independent. By assumption 1, such points exist. Point 0 will be the origin of a new reference system in which the points of the pattern will be expressed in polar coordinates (see Figure 6-1). Angle θ_j of a point j is measured with respect to the vector $p_1 - p_0$, and the orientation is such that point 2 has an angle $\theta_2 < 180^\circ$. By definition we let $\theta_0 = 0$.

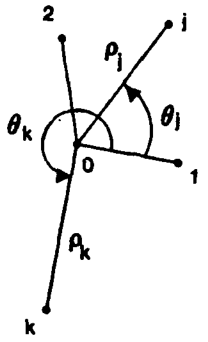
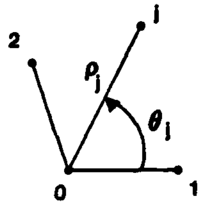


Figure 6-1: Polar coordinates in the new reference system.

In this new reference system, for all points j = 0, 1, ..., m-1 of the pattern compute their polar coordinates $\rho_j = p_{0j}$ and θ_j. We thus construct Table 6-1 with m entries each consisting of a point j and its new coordinates ρ_j and θ_j.



j	ρ_j	θ_j
0	ρ_0	θ_0
1	ρ_1	θ_1
.	.	.
.	.	.
.	.	.
m-1	ρ_{m-1}	θ_{m-1}

Table 6-1: Polar coordinates for the m points of the pattern.

6.1.4.1 The Matching Process

Consider a structure of n points 0, 1, ..., n-1. For each point of the structure do the following. Let i_0 be the point under consideration. We will try to match it to point 0 of the pattern.

A1. For i = 0, 1, ..., n-1 compute the distances from structure point i to i_0, that is, compute $\rho'_i = s_{i_0 i}$.

A2. Select those points in the structure which are at distance $p_{01} = \rho_1$ from i_0. Let

$$A = \{\, i \mid s_{i_0 i} = p_{01} \}.$$

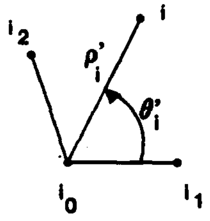
By assumption 2, |A| or the size of this set is a constant. All the points of A will be tried as matching point 1 of the pattern. Let i_1 be one point of A. Having fixed i_0 and i_1, we have at most two possible points i_2 (one being reflection of the other along $i_0\, i_1$) to match point 2 of the pattern such that

$$s_{i_0 i_2} = p_{02} \text{ and}$$

$$s_{i_1 i_2} = p_{12}.$$

The number of possible choices for (i_1, i_2) is therefore at most 2 |A|, or a constant. If A is empty or no i_2 can be found, then we conclude i_0 cannot be matched to point 0 of the pattern and leave the loop. For each choice of i_1 and i_2 do the following.

B1. Construct Table 6-2 where the n entries contain the atoms in the structure and their polar coordinates in the reference system defined by the points i_0, i_1, and i_2, in a way similar to the one shown before.



i	ρ'_i	θ'_i
0	ρ'_0	θ'_0
1	ρ'_1	θ'_1
.	.	.
.	.	.
.	.	.
n-1	ρ'_{n-1}	θ'_{n-1}

Table 6-2: Polar coordinates for the n points of the structure.

B2. We now join Table 6-1 and Table 6-2 over the compound attribute (ρ, θ). Note that because of assumption 3, this compound attribute constitutes a primary key which uniquely determines each row of Table 6-1 or 6-2. The result of this join operation will give us a list of atoms of the structure that can be matched to the atoms of the pattern.

In step B2, a complete match is found if the result table has exactly m rows. Otherwise the match is an incomplete one. Notice that this method cannot report incomplete matches when the missing atoms of the structure are those which would have been matched to points 0, 1, or 2 of the pattern.

A natural way to perform step B2 is to sort the entries of Table 6-1 lexicographically on (ρ_j, θ_j). Binary search can then be used to test whether each (ρ'_i, θ'_i) of the structure is also in the table, constructing a pair (i, j) to indicate the match in the affirmative case. The resulting algorithm is then$O(n^2 \log m)$.

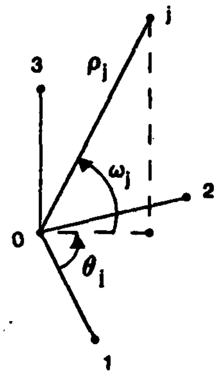


Figure 6-2: Polar coordinates in the 3-dimensional case.

In the three-dimensional case, the only difference is that we use four points to define the new reference system and that the tables have ρ, θ, ω columns. (See Figure 6-2.) Project point j onto the plane (0, 1, 2) with a direction parallel to 03. Point 0 and this projection will determine one side of the angles θ_j and ω_j. For points on the line determined by 0 and 3, this projection is defined to coincide with the point 1. The orientations of θ_j and ω_j are such that $\theta_2 < 180°$ and $\omega_3 < 180°$. During the matching process, for a given i_0 in the structure chosen to match point 0 of the pattern, the number of choices for (i_1, i_2, i_3) matching (1, 2, 3) of the pattern is still a constant, therefore the derived complexity results still apply.

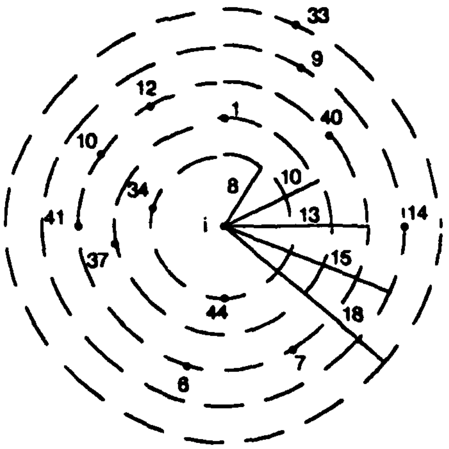
6.1.5 A Worst-Case min{$O(n^2)$, $O(n\ m^3)$} Algorithm with Preprocessing

We now describe a sequential algorithm whose worst-case time complexity is $O(n^2)$, for any distribution of points in the pattern and in the structure, given that assumptions 1 through 3 are satisfied. If the pattern points form a single conglomerate, to be defined later on, then the time complexity becomes min{$O(n^2)$, $O(n\ m^3)$}. If the pattern points are distributed uniformly in a sphere, then the algorithm will take at most $O(m\ n)$ time. Before the matching process can be started, an $O(n^2 \log n)$ preprocessing of the structure is required. This algorithm can be useful if, for a given structure, a number of different patterns are to be tested. A nice feature of this algorithm is that the matching process will be discontinued whenever it discovers a match is not possible. In the previous algorithm, having chosen one point to match point 0 of the pattern, we can conclude that a match is not possible if no points in the structure can match point 1, or points 1 and 2 of the pattern. We would like to be able to carry out the same reasoning process for each new point under consideration so that the resulting algorithm will have a nice average-case behavior.

We start with the observation that in the two-dimensional case, given three non-collinear points, any point can be determined by its distances to the three points. (In the three-dimensional case, we need four points.) Assumption 1 guarantees the existence of such points. As we will see, instead of the polar coordinates, we will define the points of the pattern and the structure by their relative distances to three selected points. A discussion of the practical usage of this algorithm is given in Section 6.1.5.4.

6.1.5.1 Structure Preprocessing

We compute all the pairwise distances among the n points of the structure and store the results in a table which allows direct access to a distance value s_{ij} given the points i and j. We then sort each row of this table in increasing order of the distance values, giving Table 6-3. An entry in row i of the this table contains the distance s_{ik} and the point k. The preprocessing phase therefore is $O(n^2 \log n)$.



row 0															
row 1															
⋮															
row i															
s_{ik}	0	8	8	10	10	13	13	13	13	13	13	15	15	18	. . .
k	i	44	34	1	37	12	6	10	40	41	7	9	14	33	. . .
⋮															
row n-1															

Table 6-3: Each entry of row i contains a sorted distance value s_{ik} and k.

6.1.5.2 Pattern Preprocessing

Let the m points of the pattern be 0, 1, ..., m-1. Again, without loss of generality, let 0 be a point whose nearest neighbor 1 and some other point 2 are such that the vectors $p_1 - p_0$ and $p_2 - p_0$ are linearly independent. All the points of the pattern will be determined by their distances to points 0, 1, and 2. For each point j, we compute p_{0j}, p_{1j}, and p_{2j}, and sort them in lexicographic order. The results are stored in Table 6-4.

Note that a number of points may be at a same distance from point 0 (and we say they all belong to a same *orbit* centered at 0). However, at most two among such points can also belong to an orbit centered at point 1. Each row of Table 6-4 also contains Level_k and $\text{NumberSatellites}_k$ (the latter being abbreviated in the table by $\#\text{Sat}_k$) for k = 0, 1, and 2. Level_0 indicates the orbit level relative to point 0 and Level_k gives the orbit level (always starting from 0) relative to point k of those atoms situated in *consecutive rows* with the same value of (Level_0, ..., Level_{k-1}). The orbit levels will be used to set up appropriate pointers to the table, so that search can be postponed and resumed without backtracking. $\text{NumberSatellites}_k$ contains a count of the number of *consecutive rows* with the same value for (Level_0, ..., Level_k). Notice that in the two-dimensional case, $\text{NumberSatellites}_1$ is at most 2 and this fact will be taken into consideration when we present the detailed algorithm in the next section. Actually we will need only Level_0 and $\text{NumberSatellites}_0$. In the three-

p_{0j}	Level_0	$\#\text{Sat}_0$	p_{1j}	Level_1	$\#\text{Sat}_1$	p_{2j}	Level_2	$\#\text{Sat}_2$	j
0	0	1	10	0	1	26	0	1	0
10	1	1	0	0	1	20	0	1	1
13	2	4	11	0	1	14	0	1	10
13	2	4	12	1	1	25	0	1	6
13	2	4	15	2	2	10	0	1	12
13	2	4	15	2	2	21	1	1	7
15	3	2	13	0	1	16	0	1	9
15	3	2	28	1	1	26	0	1	14
19	4	1	28	0	1	32	0	1	23
21	5	7	12	0	1	14	0	1	4
21	5	7	15	1	2	12	0	1	13
21	5	7	15	1	2	30	1	1	8
21	5	7	28	2	1	13	0	1	11
21	5	7	32	3	1	16	0	1	15
21	5	7	37	4	2	17	0	1	16
21	5	7	37	4	2	23	1	1	3
26	6	1	20	0	1	0	0	1	2
	⋮			⋮			⋮		⋮

Table 6-4: Distances to three selected points sorted lexicographically.

dimensional case, $\text{NumberSatellites}_2$ will be at most 2, and we will need Level_0, Level_1, $\text{NumberSatellites}_0$, and $\text{NumberSatellites}_1$.

6.1.5.3 The Matching Process

Before we present the details of the matching process, let us first show the main ideas behind it. Suppose we have already selected three points i_0, i_1, and i_2 in the structure that match points 0, 1, and 2 of the pattern, respectively. Then a point i in the structure will match a point j in the pattern if

$s_{i_0 i} = p_{0j}$,

$s_{i_1 i} = p_{1j}$, and

$s_{i_2 i} = p_{2j}$.

Thus if we had $(s_{i_0 i}, s_{i_1 i}, s_{i_2 i})$ sorted lexicographically, then we could do the matching by merely stepping through this sorted sequence and comparing with the entries of Table 6-4. However, because the choice of the points i_0, i_1 and i_2 cannot be made before we know the pattern, we cannot include this lexicographic sort in the preprocessing phase of the structure. We will show that, even without a lexicographic sort of the points in the structure, we can still carry out the matching process by stepping through rows i_0 and i_1 of Table 6-3.

We initialize a pointer (which we call StructurePointer) to the beginning of row i_0 of Table 6-3 and another (which we call PatternPointer) to the beginning of Table 6-4. We advance StructurePointer along row i_0 until the current distance value $s_{i_0 i}$ is equal to the current distance value p_{0j} pointed to by PatternPointer. Denote this common length by d. (If $s_{i_0 i} > p_{0j}$ then we have a mismatch and the process can be discontinued.) The number of atoms in subsequent entries of Table 6-4 which are also at distance d from point 0 is given by NumberSatellites$_0$. Denote by B the set of atoms in the structure at distance d from i_0. The size of B is at least 1 since point i is such a point. If NumberSatellites$_0$ = 1 then we scan through B and either one of its members matches atom j of the pattern, in which case we advance both PatternPointer and StructurePointer until they point to entries with distance values different from d, or else we discontinue the matching process. If NumberSatellites$_0$ is greater than |B| then we conclude that a match is not possible and discontinue the process.

The processing becomes more complicated if NumberSatellites$_0$ is greater than 1 but not greater than |B|. The decision of whether such atoms in the pattern can be matched by members of B will be postponed. The current value of PatternPointer will be saved for future use. We use ArrayPatternPointer[0:m-1] and ArrayLevel[0:n-1] and define some of their elements as follows. We set ArrayPatternPointer[Level$_0$] to PatternPointer and we set ArrayLevel[k] to Level$_0$ for all k in B. (See Figure 6-3 for an example where, for readability, points in the structure and pattern that match have the same name and points in rows i_0 and i_1 that do not match any point in the pattern are represented by dark square dots.) Both PatternPointer and StructurePointer will be advanced to their respective new entries with a distance value different from d. Having thus scanned row i_0 (or having reached the end of Table 6-4), we then step through row i_1. We need to consider only those entries ($s_{i_1 i}$, i) for which ArrayLevel[i] has been defined in the previous pass. For each of such point i ArrayPatternPointer[ArrayLevel[i]] will play the role of the current PatternPointer. As the

Table 6-3

row i_0														. . .												
$s_{i_0 i}$	0	8	8	10	10	13	13	13	13	13	13	15	15	18	.19	19	21	21	21	21	21	21	21	21	26	26	. . .
i	i_0	■	■	i_1	■	12	6	10	■	■	7	9	14	■	■	23	■	13	4	8	11	3	15	16	■	2	. . .
														. . .												
row i_1																										
$s_{i_1 i}$	0	7	10	11	11	11	12	12	13	13	15	15	15	15	20	28	28	28	28	28	30	32	32	37	37	37	. . .
i	i_1	■	i_0	■	■	10	4	6	9	■	12	8	7	13	2	23	■	14	11	■	■	·15	■	3	16	■	. . .
														. . .												

ArrayLevel	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]	[16]	. . .	[n-1]
				5	5		2	2	5	3	2	5	2	5	3	5	5	. . .	

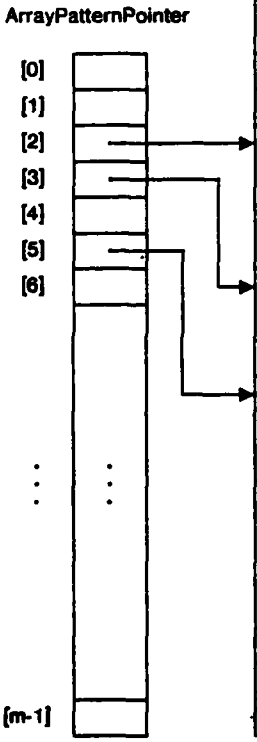


Table 6-4

p_{0j}	$Level_0$	$\#Sat_0$	p_{1j}	$Level_1$	$\#Sat_1$	p_{2j}	$Level_2$	$\#Sat_2$	j
0	0	1	10	0	1	26	0	1	0
10	1	1	0	0	1	20	0	1	1
13	2	4	11	0	1	14	0	1	10
13	2	4	12	1	1	25	0	1	6
13	2	4	15	2	2	10	0	1	12
13	2	4	15	2	2	21	1	1	7
15	3	2	13	0	1	16	0	1	9
15	3	2	28	1	1	26	0	1	14
19	4	1	28	0	1	32	0	1	23
21	5	7	12	0	1	14	0	1	4
21	5	7	15	1	2	12	0	1	13
21	5	7	15	1	2	30	1	1	8
21	5	7	28	2	1	13	0	1	11
21	5	7	32	3	1	16	0	1	15
21	5	7	37	4	2	17	0	1	16
21	5	7	37	4	2	23	1	1	3
26	6	1	20	0	1	0	0	1	2
	⋮			⋮			⋮		⋮

Figure 6-3: *An example showing the matching process.*

StructurePointer steps through row i_1, different elements of ArrayPatternPointer will be used. However, the important fact is that each individual ArrayPatternPointer[i] only advances and no backtracking is ever needed. This important behavior guarantees the matching process to be linear in n in the worst case.

We now present the algorithm in more detail. We will use ArrayMatch[0:m-1] where ArrayMatch[j] will contain the structure atom that matches atom j of the pattern. A complete match is obtained when ArrayMatch is totally defined. A test that this has occured should be made whenever ArrayMatch is updated. Such tests will be omitted in the presentation that follows, for the sake of readability. Recall that we also have a table which allows direct access to a distance value s_{ij} given two points i and j of the structure.

For each point in the structure do the following. (Let i_0 be the point under consideration. We will try to match it to point 0 of the pattern.)

A1. Consider row i_0 of Table 6-3 and obtain a set A of points of the structure at distance p_{01} from i_0. Each point of A will be tried as matching point 1 of the pattern. As in the previous algorithm, the size of A is a constant and, for a given point i_1 of this set, we can have at most two points i_2 which match point 2 of the pattern. This guarantees the number of choices of (i_1, i_2) to be at most 2 |A|, or a constant. If no i_1 or i_2 can be found, we discontinue the matching process. (Whenever the matching process is to be discontinued because match is not possible or a complete match has been found, we return to this point.) For each new choice of i_1 and i_2 do the following.

B1. [Scanning row i_0 of Table 6-3.] Initialize all elements of ArrayMatch, ArrayLevel, and ArrayPatternPointer to null. Set StructurePointer to the beginning of row i_0 of Table 6-3 and PatternPointer to the beginning of Table 6-4. Let i and j denote the atoms in the current entries pointed to by StructurePointer and PatternPointer, respectively. While both pointers are within valid range do the following.

We have one of the following three cases.

C1. $s_{i_0 i} < p_{0j}$. Advance StructurePointer.

C2. $s_{i_0 i} = p_{0j} = d$.

There are three cases to consider where, in cases D2 and D3, let $B = \{ k \mid s_{i_0 k} = d \}$.

D1. $\text{NumberSatellites}_0 = 1$.

If $s_{i_1 i} > p_{1j}$ then match is not possible

else if $s_{i_1 i} = p_{1j}$ and $s_{i_2 i} = p_{2j}$

then set ArrayMatch[j] to i and advance both PatternPointer and StructurePointer

else advance StructurePointer.

D2. $1 \leq |B| < \mathrm{NumberSatellites}_0$. Match is not possible.

D3. $1 < \mathrm{NumberSatellites}_0 \leq |B|$. Set ArrayPatternPointer[Level_0] to PatternPointer. Set ArrayLevel[k] to Level_0 for all k in B. Advance PatternPointer (resp. StructurePointer) until p_{0j} (resp. $s_{i_0 i}$) > d or pointer out of valid range.

C3. $s_{i_0 i} > p_{0j}$. Match is not possible.

B2. [PatternPointer or StructurePointer out of valid range.] If PatternPointer within valid range then match is not possible.

B3. [Scanning row i_1.] Initialize StructurePointer to the beginning of row i_1. Let i denote the atom in the current entry pointed to by StructurePointer. Let j denote the atom in the current entry of Table 6-4 pointed to by ArrayPatternPointer[ArrayLevel[i]], if ArrayLevel[i] is not null. We say that ArrayPatternPointer[k] is valid if it is not null and points to a current entry inside Table 6-4 with Level_0 = k. While StructurePointer is within valid range and there exists some k for which ArrayPatternPointer[k] is valid do the following.

If ArrayLevel[i] = null or ArrayPatternPointer[ArrayLevel[i]] is not valid then advance StructurePointer else we have one of three cases:

C1. $s_{i_1 i} < p_{1j}$. Advance StructurePointer.

C2. $s_{i_1 i} = p_{1j}$. There are four possibles cases:

D1. $s_{i_2 i} < p_{2j}$. Advance StructurePointer.

D2. $s_{i_2 i} = p_{2j}$.

Set ArrayMatch[j] to i. Advance ArrayPatternPointer[ArrayLevel[i]]. If ArrayMatch[j] is not null and ArrayPatternPointer[ArrayLevel[i]] is valid advance ArrayPatternPointer[ArrayLevel[i]]. Advance StructurePointer.

D3. $s_{i_2 i} > p_{2j}$ and $s_{i_2 i} = p_{2k}$,

where k is the atom in the next entry relative to ArrayPatternPointer[ArrayLevel[i]]. Set ArrayMatch[k] to i. Do *not* advance ArrayPatternPointer[ArrayLevel[i]] because atom j still needs to be matched. Advance StructurePointer.

D4. $s_{i_2 i} > p_{2j}$ and $s_{i_2 i} \neq p_{2k}$,

where k is the atom in the next entry relative to ArrayPatternPointer[ArrayLevel[i]]. Match is not possible.

C3. $s_{i_1 i} > p_{1j}$. Match is not possible.

B4. [Row i_1 has been processed or all elements of ArrayPatternPointer are invalid.] If there is still some valid element of ArrayPatternPointer then match is not possible.

In the three-dimensional case, the points will be expressed by their distances to four selected points. The scanning of row i_0 will be very similar to what we have above. During the scanning of row i_1, let j be the current entry pointed to by the current pattern pointer, as determined by the current entry i of row i_1. We have $s_{i_0 i} = p_{0j}$. Entry j may have a value of $\text{NumberSatellites}_1$ no longer bounded by 2. In that case, if $s_{i_1 i} = p_{1j}$, we again save the current pattern pointer in a *two-dimensional* array indexed by Level_0 and Level_1. The values of the *two* indices will be stored in ArrayLevel[i]. (Other points k for which ArrayLevel[k] will also receive the same content need not occur in a contiguous portion of row i_1). In addition to rows i_0 and i_1, row i_2 may also have to be scanned. The processing of row i_2 is similar to that of row i_1 in the two-dimensional case. We will not elaborate the details here.

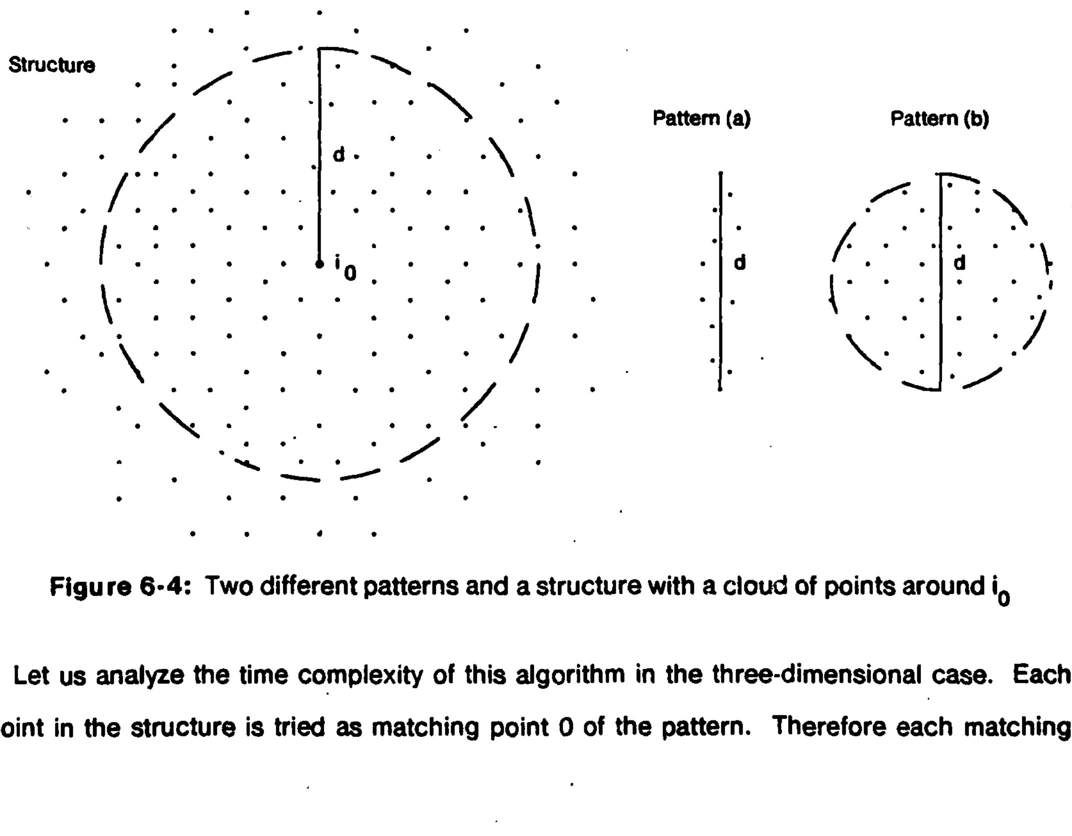


Figure 6-4: Two different patterns and a structure with a cloud of points around i_0

Let us analyze the time complexity of this algorithm in the three-dimensional case. Each point in the structure is tried as matching point 0 of the pattern. Therefore each matching pass, *from* steps B1 *through* B4, *is executed* O(n) *times.* The time to execute one matching pass is, in the worst case, proportional to the time to step through the initial elements of rows i_0, i_1, and i_2, where the distance values are not greater than the diameter of the pattern. It is instructive to look at the examples in Figure 6-4. Let d be the diameter of the pattern. In the worst case, all the points inside the sphere with center at i_0 and radius equal to d will be examined in a matching pass. The number of such points depends on the length d and on the distribution of the points. It is obviously less than n, which puts an upper bound of O(n) on one matching pass. Thus the algorithm takes at most $O(n^2)$ time. Let us now derive another bound, in terms of m or the number of points in the pattern. Let us first give the definition of what we mean by *single conglomerate.* Consider the distances between each of a collection of points and its nearest neighbor. Let d_{min} and d_{max} denote the minimum and maximum of such distances. Construct an undirected graph G from the collection of points as follows. The vertices of graph G are the points of the collection. If the distance between points i and j is less or equal to d_{max}, then connect the corresponding vertices in G with an edge. We say that the collection of points forms a *single conglomerate* if

(i) $d_{max} \leq c\, d_{min}$, for some small constant c independent of n, and

(ii) the graph G as constructed above is connected.

(A graph is connected if there exists a path between any two of its vertices.) Assume that the points of the pattern constitute a single conglomerate. The pattern will give a maximum diameter of length O(m), when its points are distributed in the vicinity of a straight line (as in pattern (a) of Figure 6-4). The maximum number of points in the structure that need to be examined is $O(m^3)$, when there is a cluster of points around i_0. Hence the worst-case time complexity for the proposed algorithm is $\min\{O(n^2), O(n\, m^3)\}$. The points of pattern (b) in Figure 6-4 are uniformly distributed in a sphere. For such a pattern, a matching pass will take at most O(m) time, thus giving a total time of at most O(m n).

The matching process can be viewed as a number of join operations. Consider the set of distance values of points of the pattern to point 0

$$D = \{\, d \mid d = p_{0j} \text{ for some point } j \text{ of the pattern} \},$$

and let the elements of this set be

$$d_0 < d_1 < \dots < d_{|D|-1}.$$

We can partition Table 6-4 into smaller tables

$$P_k = \{ (p_{0j}, p_{1j}, p_{2j}, j) \mid p_{0j} = d_k \}, \text{ for } k = 0, 1, ..., |D|\text{-}1.$$

Therefore, having chosen (i_0, i_1, i_2), the matching process will consist of obtaining each table

$$S_k = \{ (s_{i_0 i}, s_{i_1 i}, s_{i_2 i}, i) \mid s_{i_0 i} = d_k \}$$

and joining it with table P_k over the compound attribute defined by the first three columns, for $k = 0, 1, ..., |D|\text{-}1$. Having computed one join operation we will proceed to the next one only if all the points in P_k have been matched to points in S_k. Otherwise the match under consideration is impossible.

6.1.5.4 Practical Considerations

Suppose we are given a pattern with a large diameter. Consider points in the pattern which are very far away from point 0 and its nearest neighbors 1 and 2. Such far away points, when expressed in terms of their distances relative to 0, 1, 2, and 3, may be hard to distinguish one from the other, since three of the distance values may be nearly equal. In this case, we may prefer to use the first algorithm (or some variation of that algorithm, with the same kind of preprocessing of the structure so as to achieve good average-case performance). For patterns with small diameters, however, this does not pose a problem even though the structure has a large diameter, because we really never go beyond the diameter of the pattern in our matching process.

Concerning the implementation of the proposed algorithm, one immediate observation is that instead of a distance value d, we can use instead d^2, thus saving us the time to extract square roots. This observation also applies to the previous algorithm. Another observation is that certain entries in the various tables may be chained in such a way that consecutive entries with same distance values may be skipped in a single step.

6.2 Three-Dimensional Shape Matching

Consider a three-dimensional *rectilinear shape* as any finite arrangement of straight line segments (possibly with zero length) in a finite volume. In the following we consider rectilinear shapes. Two shapes are *similar* if one of the two coincides with the other after some Euclidean transformations that include translation, rotation, mirror image, and scale, or a finite composition of these. One shape is a *subshape* of a second if every part of the first shape is also a part of the second. Given three-dimensional rectilinear shapes p and s, we wish to report all subshapes of s which are similar to p. An example is shown in Figure 6-5.

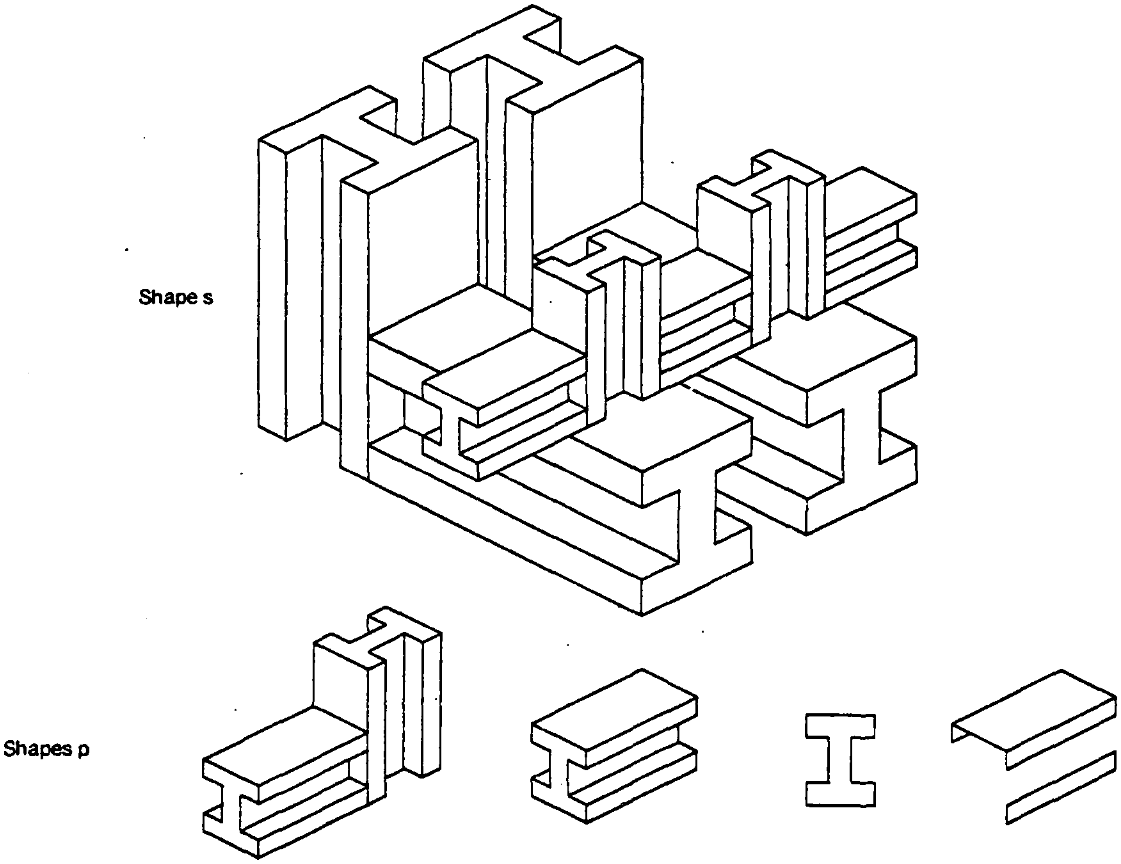


Figure 6-5: Shape matching examples in 3-space.

One important application of this problem is in formal composition of component building systems. Shape grammars have been proposed for characterizing formal composition of shapes [70, 86, 87], where the shape matching problem as stated above constitutes an important step in the application of the production rules. In the two-dimensional case, this problem is important for example in floor plan composition [88]. It may also be of some interest in certain databases containing layouts of components to extract subparts of certain shapes.

6.2.1 Relation to the Previous Problem and New Assumptions

The previous matching problem can be viewed as a special case of the three-dimensional shape matching problem, when all the line segments have zero length, and scale is not included in the set of allowable Euclidean transformations. Our goal here is to solve the three-dimensional shape matching problem by modifying the previous algorithm without however increasing its complexity. In order to be able to use portions of the previous algorithm, we will use the same names as before. If we wish to report all subshapes of shape s that are similar to shape p, we will denote *shape s* and *p* by the names of *structure* and *pattern*, respectively. In addition to the Cartesian coordinates (and possibly some attribute values) of the n points of the structure and m points of the pattern, we are also given the set of points that are connected to each point. A necessary condition for a pair of points in the structure to match a pair of points in the pattern that are connected is that they too be connected. Notice that the pattern (0, 1, 2) in Figure 6-6 does not match (A,D,H) nor (E,A,D) because DH and AE are not connected. However, in order to match two points in the pattern that are not connected, it is not necessary for the matching points in the structure to be also disconnected.

For this problem, we say that point A is a "neighbor" of point B if A and B are connected. Thus assumptions G1 through G4 of Section 6.1.3 can be re-stated as follows.

1'. There exist points j_0, j_1, j_2, and j_3 in the pattern, with point j_0 connected to points j_1 and j_2, such that

$$p_{j_1} - p_{j_0},\ p_{j_2} - p_{j_0},\ \text{and } p_{j_3} - p_{j_0}$$

are linearly independent. (In the two-dimensional case, we need three points j_0, j_1, and j_2, with j_0 connected to j_1, such that $p_{j_1} - p_{j_0}$ and $p_{j_2} - p_{j_0}$ are linearly independent.)

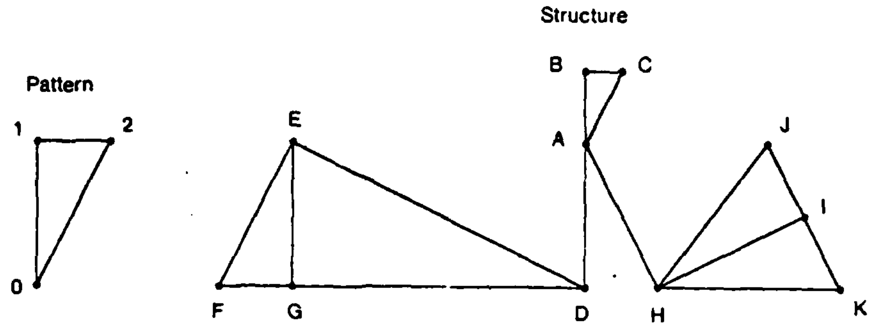


Figure 6-6: (0,1,2) matches (A,B,C), (D,E,F), (D,G,E), (E,G,F), (H,I,J), (H,I,K)

2'. The connectivity of any point in the pattern or in the structure is bounded by a constant.

3'. Same as G3.

4'. Same as G4.

With an adequate choice of the points j_0, j_1, j_2, and j_3, it may be possible to represent all the points in the pattern by their distances to these points, with no precision concerns as mentioned earlier in Section 6.1.5.4.

Under these assumptions, we now present a modified algorithm for the three-dimensional shape matching problem, with a worst-case time complexity of $O(n^2)$ and preprocessing of $O(n^2 \log n)$.

6.2.2 The Modified Algorithm

We consider directly the three-dimensional case. Both the structure and the pattern will be preprocessed in the same way as before. Without loss of generality, we again assume that points 0, 1, 2, and 3 of the pattern have the property of the points j_0, j_1, j_2, and j_3, respectively, of assumption 1'.

There are three main differences in the matching process. The first difference is in step A1, Section 6.1.5.3. For each point in the structure chosen as point i_0 to match point 0 of the pattern, all the points connected to i_0 will be tried as matching point 1 of the pattern. By assumption 2', the number of such points is a constant. For a given choice of (i_0, i_1) the number of points i_2 that can match point 2 of the pattern is again a constant, since i_2 must be connected to i_0. Finally, for a given choice of (i_0, i_1, i_2), there are at most two points i_3 eligible to match point 3 of the pattern. This ensures that for each point i_0 chosen to match point 0 of the pattern, steps B1 through B4 will be executed, not necessarily to its entirety, a constant number of times.

Let q denote the quotient between $s_{i_0 i_1}$ and p_{01}. The second difference is that all distance values in the pattern must first be multiplied by q before they can be used in comparisons with distance values in the structure.

Finally, the third difference is that each time ArrayMatch is updated, say ArrayMatch[j] is to be set to i, we have to test ArrayMatch[j_p], where j_p are points in the pattern connected to j. Either ArrayMatch[j_p] is null or it should be equal to a point connected to i. This test will ensure that the connectivity in the pattern is retained in the matching subshape in the structure.

The above modifications do not alter the worst-case time complexity which is still $O(n^2)$. However, we are no longer able to give a bound in terms of m. The inclusion of scale in the set of allowable Euclidean transformations makes the matching process more complex. If the pattern consists entirely of isolated points, then for each point i_0 in the structure chosen to match point 0 of the pattern, the number of possible values of (i_1, i_2) that need to be tested for match with points 1 and 2 of the pattern would no longer be a constant. Nevertheless, we have managed to produce a modified algorithm for the three-dimensional shape matching problem with the same complexity as the previous algorithm, where scale was not included. We have traded off generality for performance. The complexity did not increase because of the assumptions of bounded connectivity (assumption 2') and the existence of the four reference points in the pattern with certain special characteristics (assumption 1'). Concerning the latter assumption, we observe that the only connectivity requirement is that point i_0 be connected to j_1 and j_2. All the remaining points in the pattern may be isolated points, with zero connectivity. For example, the matching problem (with scale included) as illustrated in Figure 6-7 where all but three points in the pattern have zero connectivity, satisfies assumptions 1' through 4' and therefore the modified algorithm is applicable. Notice the shapes in this example are the same as in Figure 6-5, except that many lines have been removed.

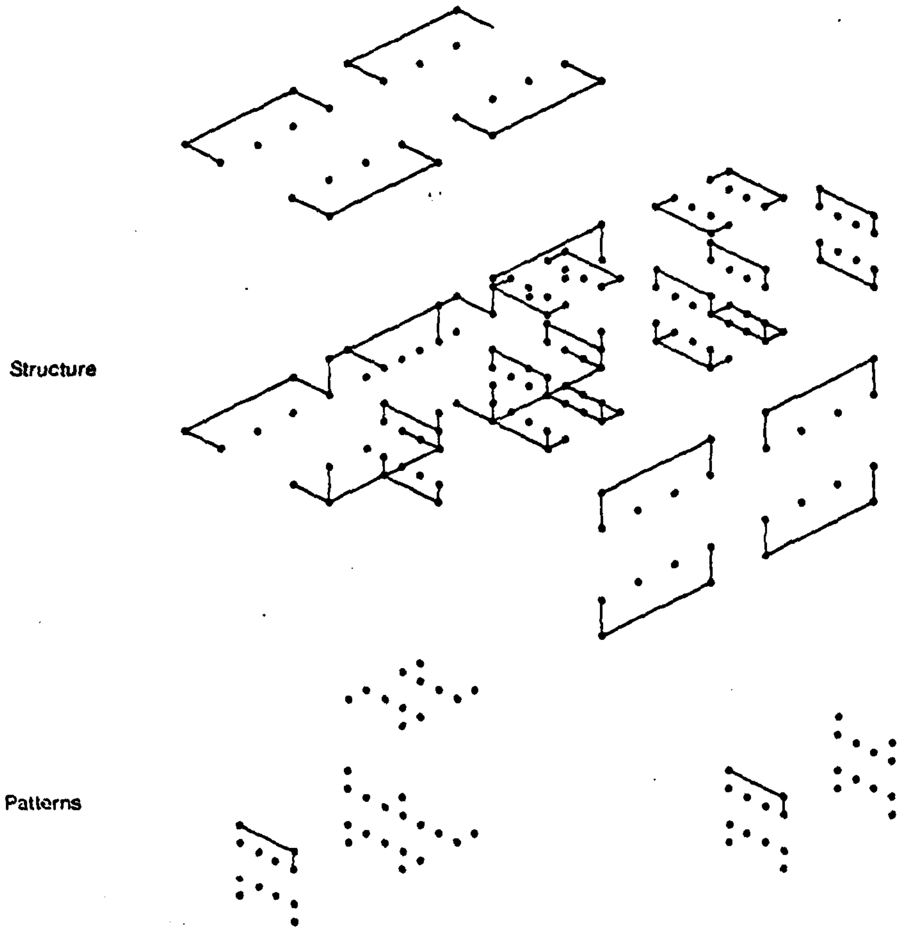


Figure 6-7: Patterns consisting mostly of points.

6.3 Reporting Pairwise Intersections of n Rectangles

Consider the problem of detecting and reporting all occurrences of pairwise intersections of a collection of n rectangles. One application is design-rule checking in a database containing descriptions of VLSI layouts. The objects are usually the components on a chip or their bounding boxes. The reported intersections can be used to pinpoint potential trouble areas for further processing. Many design rules such as minimum width and clearance can be performed by a combination of programs to expand/shrink rectangles, perform logical operation, followed by an intersection reporter [33].

One important characteristic in the distribution of objects in such a database is that of locality. Objects representing VLSI components tend to distribute themselves uniformly over a chip's bounding box. As a result, one object will intersect a few neighbor objects. This means that the naive algorithm of checking all $O(n^2)$ pairs of objects for intersection is not appropriate. Bentley, Haken, and Hon [8] confirm this assumption by studying a number of specific Mead-Conway style designs. Furthermore, the authors give *a priori* arguments, based on VLSI design philosophy, that justify the uniform distribution assumption. The aim of making effective use of the silicon area implies that large blank spaces tend to be rare in good designs. On the other hand, the number of objects that cluster around a given point is limited because of the fixed number of layers available and the design practice of not overlapping many objects on the same layer. The locality property has long been recognized and exploited in earlier design rule checking systems such as [4] and [94]. A recent work [8] presents a linear expected time algorithm to solve the rectangle intersection problem, with a preprocessing step consisting of a sort of the rectangles. The analysis of the derived algorithm is done under a formal probabilistic model.

We will briefly describe the main ideas of algorithms that take advantage of locality and then show that they can be carried out very nicely with a special-purpose device capable of maintaining a collection of objects with the possible operations of insertion, deletion, and search. We then present a decomposition procedure that handles the case when the problem size exceeds that of the special-purpose device. We will show that because of the locality property, the device size can be quite small.

6.3.1 Main Ideas

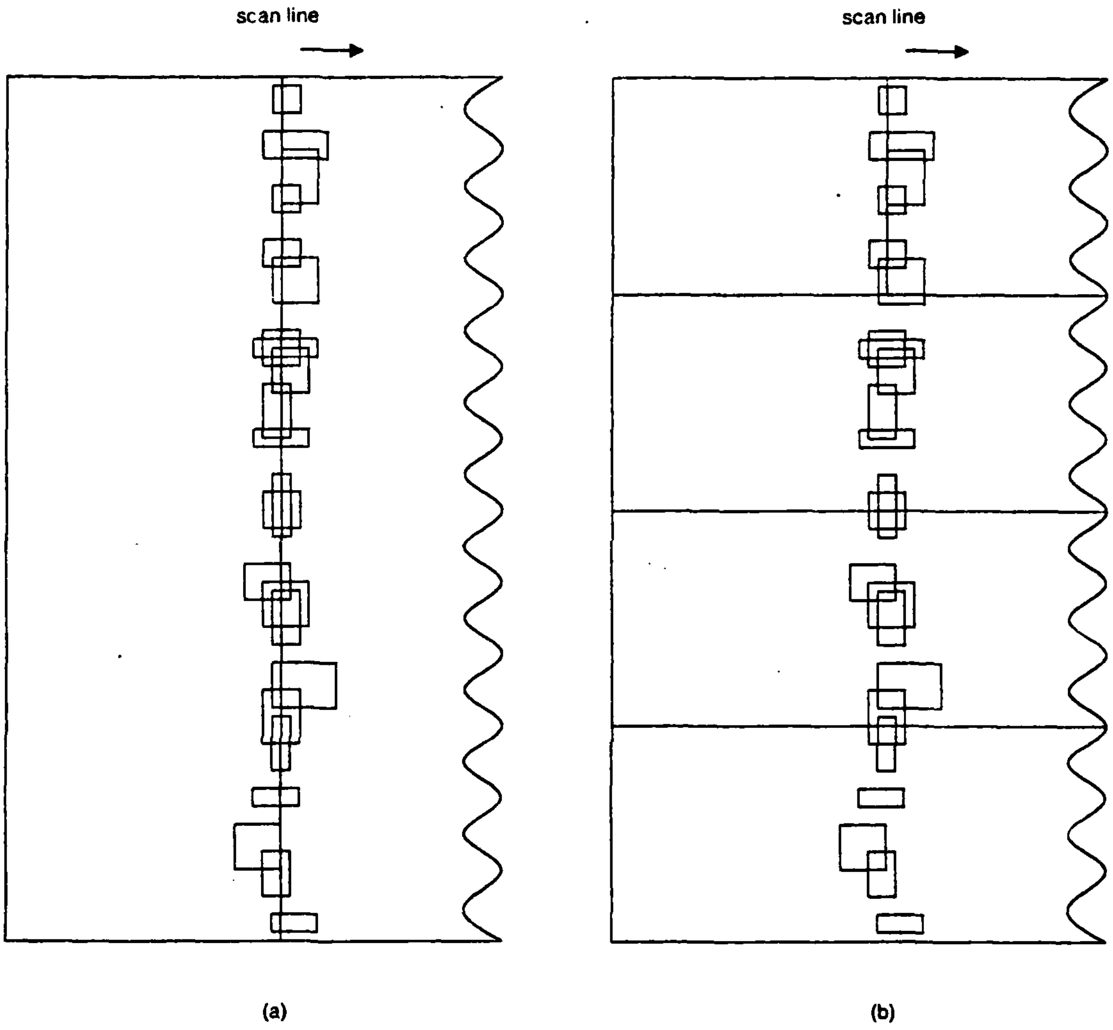


Figure 6-8: Use of a scan line to determine the active set.

Each rectangle has its left and right edges represented in a *vertical-edge sequence* where they are sorted in increasing order by the x-coordinate. The vertical-edge sequence will indicate the sequence of vertical edges touched by a vertical scan line sweeping the rectangles from left to right, as illustrated in Figure 6-8 (a).

Let *active set* denote the set of rectangles kept in main memory and *conflict set* denote a subset of the active set where search for intersection is performed. Starting from an empty active set and the scan line at the left edge of the chip bounding box, move the scan line toward the right. Each time the scan line touches a left vertical edge of a new rectangle, the corresponding object is entered to the active set. Each time a right vertical edge is detected, the corresponding object is removed from the active set. For a given position of the scan line, the active set will contain the rectangles intersecting the scan line. The above procedure can be carried out by stepping through vertical-edge sequence and updating the active set accordingly.

A detailed study of the statistics of shapes of a sample of designs was made in [8]. Rectangles were classified into three categories: components (neither side greater than 10 λ), wires (one side greater than 10 λ and the other not greater than 6 λ), and others. It was found that components and wires constitute about 98% of the total (with roughly 73% of components and 25% of wires). If d denotes the mean edge length of all the rectangles, then about 99% of the rectangles have the longer edge less than 20d. (d was found to be approximately 7.6 λ.) It was also found that n rectangles on a chip are distributed uniformly over a bounding box which is a square of side $d'n^{1/2}$. (d' was found to be 8 λ, a little longer than d.)

In the proposed algorithm the conflict set size is a constant. The algorithm needs $O(n^{1/2})$ space for the storage of the active set and O(n) time for detecting and reporting the intersections, excluding the initial sorting phase (an external sort). Notice that this method can be extended to handle the detection of pairwise intersections of objects other than rectangles. It suffices to consider the bounding boxes of the objects and, instead of reporting an intersection of two bounding boxes, the algorithm checks if the two corresponding objects in fact intersect.

6.3.2 A Partitioning Scheme

One straightforward way to use a special-purpose device is to use it to implement the active set. The device will maintain the collection of rectangles that intersect the current position of the scan line. The operations needed are insertion, deletion, and search. Recall that the size of the active set is $O(n^{1/2})$. The locality property suggests a natural decomposition procedure to make this size constant.

First we repeat an argument used in [8] to prove the linearity of the algorithm. Under the proposed probabilistic model, the n rectangles are squares of edge length d distributed uniformly over a square of side $d'n^{1/2}$. The active set corresponds to the set of rectangles intersected by the current scan line. The probability that a given rectangle intersects a given scan line is $d/(d'n^{1/2})$. The expected number of rectangles intersecting the scan line at any given time is n times this probability or $d\ n^{1/2}/d'$, which is roughly $n^{1/2}$. Divide the chip bounding box (of side $d'n^{1/2}$) into $n^{1/2}$ horizontal partitions. Corresponding to each partition *one bin is created* whose usage is as follows. If a new left edge intersects the current scan line, its name is inserted into all bins corresponding to the partitions spanned by the rectangle in question. Each rectangle in the active set is entered into two (or possibly one) bins. Since rectangles are uniformly distributed, each bin contains approximately two rectangles in the active set.

Let us assume that, instead of $n^{1/2}$ partitions, the chip bounding box is divided into $n^{1/2}/k$ partitions, for some integer k. (See Figure 6-8 (b).) Then, by similar reasoning, each partition will contain about $k+1$ rectangles in the active set. This will suggest the decomposition procedure. The height of one partition is

$$d'\ n^{1/2} \ / \ (n^{1/2} \ / \ k) = d'\ k.$$

The value of k will be chosen such that the probability of a rectangle spanning three or more partitions is nearly zero. Using the data from [8], which shows that 99% of the rectangles have the longer edge less than 20d, we have

$$d'\ k > 20d, \text{ or } k > 20\ d/d' \cong 19.$$

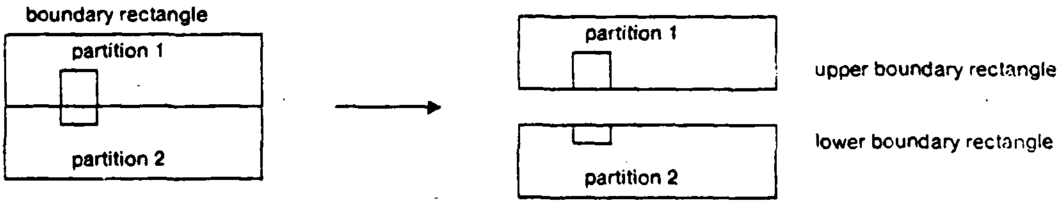


Figure 6-9: A boundary rectangle.

The scan line now sweeps each partition in turn. We need to resolve cases in which the rectangles cross partition boundary lines and lie in more than one partition. Consider rectangles that lie in two partitions. (See Figure 6-9.) Let us call these by the name of *boundary rectangles*. Denote by *upper boundary rectangle* the part of the boundary rectangle that lies above a boundary line and by *lower boundary rectangle* the one below the boundary line. Since a boundary rectangle is split into two, we would like to avoid reporting twice the intersection of two boundary rectangles crossing the same boundary line. This can be achieved by not reporting intersection of two lower boundary rectangles.

The number of partition boundary lines is $n^{1/2}/k - 1$. Instead of n, the total number of rectangles to be processed is

$$n + (n^{1/2}/k - 1)\, n^{1/2} < n + n/k = (1 + 1/k)\, n.$$

With this decomposition procedure, instead of solving one large problem with n rectangles, we now solve $n^{1/2}/k$ subproblems each consisting of $(1 + 1/k)n/(n^{1/2}/k)$, or roughly $k\ n^{1/2}$ rectangles. Each subproblem consists of the sorting of $k\ n^{1/2}$ rectangles, and the scan line now intersects about k rectangles.

To provide some safety margin, we can use a special-purpose device of size say 4 or 5 times k. For $k = 20$, a device of size around 100 should be appropriate. An overflow area can be maintained in main memory in case the active set size exceeds that of the special-purpose device. Search of rectangles that intersect in the overflow area will have to be done sequentially. Occurrence of overflow should be very rare, so that the overall performance is not affected.

One immediate advantage of the decomposition scheme is that the sorting phase involves a much smaller quantity of rectangles. External sorting may not be necessary altogether. Another advantage is that the number of rectangles intersected by the scan line is now a constant. This means less storage demand, as well as easier book-keeping operations. With the decomposition procedure, the active set is equal to the conflict set, both of constant size. This is a desirable feature in a design rule checking algorithm especially when the component count on a chip continues to increase. (Another desirable feature is to take advantage of the high-level information in hierarchical designs [10, 36, 93], a topic that falls outside the scope of the present work.)

6.4 A Multidimensional Search Problem

Consider the following multidimensional search problem. Given n lines in a k-dimensional space, where at most a constant number of lines intersect at a same point, answer if a given query point lies on any of the lines and, in the affirmative case, identify all such lines. Many variations of this problem are presented in [26], where the above problem in the two-dimensional case is solved with a query time of O(log n) at the cost of an expensive preprocessing.

One useful application of this problem or some of its variants, in the two-dimensional case, is in geographic or cartographic databases. In [47] it is argued that, as an alternative to the grid subdivision method, the polygonal subdivision method constitutes a better way of representing geographic objects, though much more complicated data need to be stored and processed. The polygonal subdivision representation has been employed in several geographic databases for the processing Metropolitan maps. These systems treat roads as line segments and a road network is expressed by a set of line segments plus some other information. The condition of the problem is easily met since only a few (usually two or three) roads intersect at the same point. A geographic database system has been implemented under the relational model where the basic entity is a map which is a collection of points, lines, line groups, and zones [12].

6.4.1 Related Work

Dobkin and Lipton [26] present an algorithm based on a generalized binary search which gives a fast O(log n) query time in the two-dimensional case. Since lines in the plane are not ordered in any obvious way, the proposed algorithm relies on an expensive preprocessing of the n lines which is O(n^3 log n) time and O(n^3) space.

Lee and Preparata [61] consider the planar subdivision problem, a variant of this problem, which can be solved in O(log m) time with preprocessing time O(m log m) where m can be viewed as the number of intersections of the given line segments. Since O(n^2) intersections are possible, the preprocessing can still be very expensive.

Another variant is when a batch of n known points are to be queried, as opposed to the on-line problem where the query points are not known in advance. Shamos and Hoey [81] present an O(n log n) algorithm which answers if there exists some query point that lies on the given lines, without giving individual answers concerning which points lie on which lines.

6.4.2 A Hardware-assisted Solution

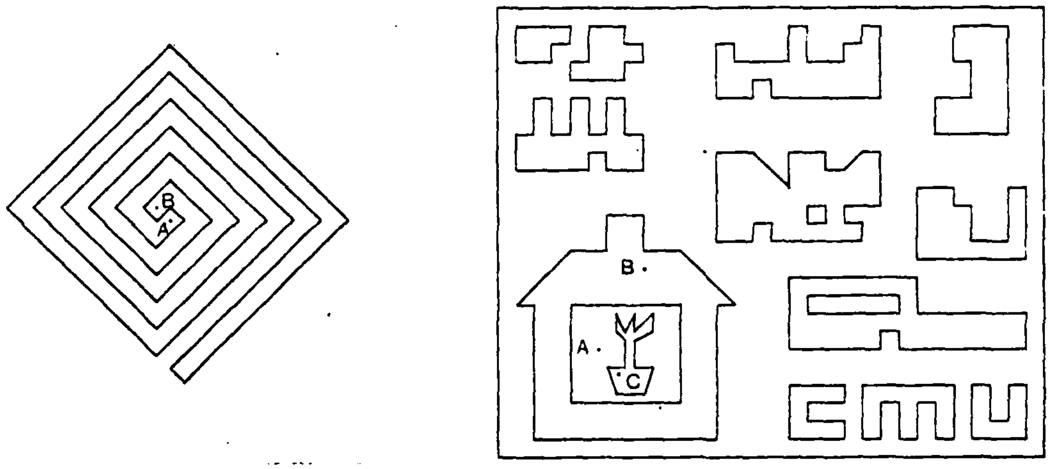
We consider the two-dimensional case and examine two problems. In the first one, a next query point will be given only after the answer to the current query has been supplied. A straightforward hardware-oriented solution is to use a tree device of capacity s $\geq$ n to hold all n given lines. The "preprocessing" now consists of loading the special-purpose device with parameters that define the n line segments. A given query point is broadcast to all the leaves where tests are made to see if it lies on any of the given lines. Since at most a constant number of lines can intersect at a given point (assumption of the problem), this will result in an O(log s) algorithm. This on-line problem does not seem to yield any decomposition procedure in case s is less than n. Any partitioning mechanism would result in a trivial O(n) solution, which is of course not interesting.

In the second problem a list of m query points are given one by one (in an on-line fashion) and in return we have to supply a list of the corresponding answers. The case n $\leq$ s can be handled by first loading the n lines into the special-purpose device. Then each of the m query points is broadcast to the leaves where comparisons take place. By means of pipelining and given that one query point can lie on at most a constant number of lines, the total query time will be O(m). In case multiple results need to be output the flow of data in the various kinds of internal nodes should be disciplined, as described in Chapter 3.

If n > s, then we can carry out the same procedure n/s times, each time loading s lines into the device and processing the m query points. The total time will be O(m n/s). This represents a speed up of O(s) with respect to the trivial O(m n) sequential algorithm, which we would use if we do not want to invest in an expensive preprocessing.

6.5 The Containment Problem

Consider the following containment problem. Given a simple polygon of n edges, answer if each of a sequence of m query points lies inside the polygon. Figure 6-10 (a) illustrates an example where point A lies inside the given polygon while point B does not.



(a) (b)

Figure 6-10: The 2-D containment problem and one possible extension.

The 2-D containment problem is important in computer graphics [76]. Together with its 3-D counterpart, it constitutes a fundamental problem in design databases for architectural or mechanical design purposes. Kalay and Eastman [42] discuss shape operations such as union, intersection, and difference, where the 2-D and 3-D containment tests constitute one main part of their algorithm. The 2-D containment problem is also discussed in [7] and [82].

One possible generalization of this problem is to allow "holes" and "islands" in the polygon. It can be used to describe the contour of objects in a planar layout (see Figure 6-10 (b)). In this example, the "holes" can be viewed as representing contours of objects. Given m points, we want to know if each point lies inside a non-empty space, which may indicate a spatial conflict. In the example, point A is inside an empty space while B and C are not.

One often used method is illustrated in Figure 6-11 (a) and (b). Point P is inside the polygon

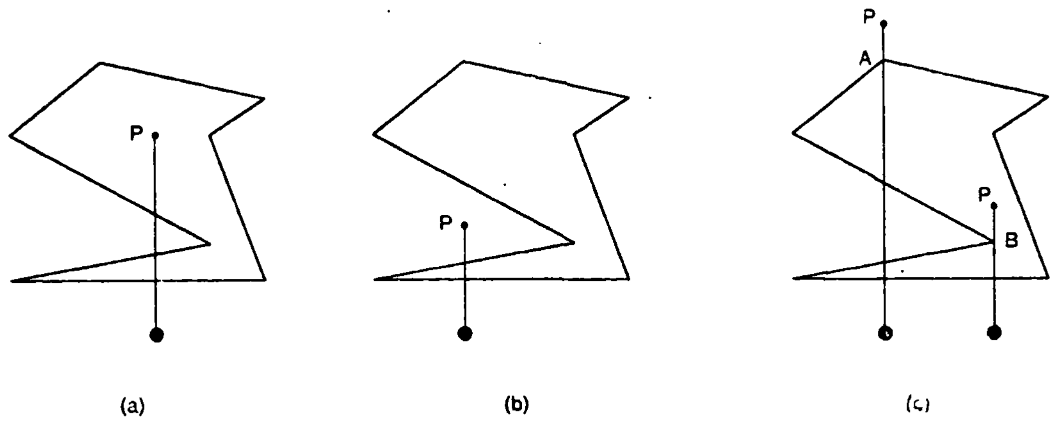


Figure 6-11: Counting the number of edges above the query point; problem with singularity.

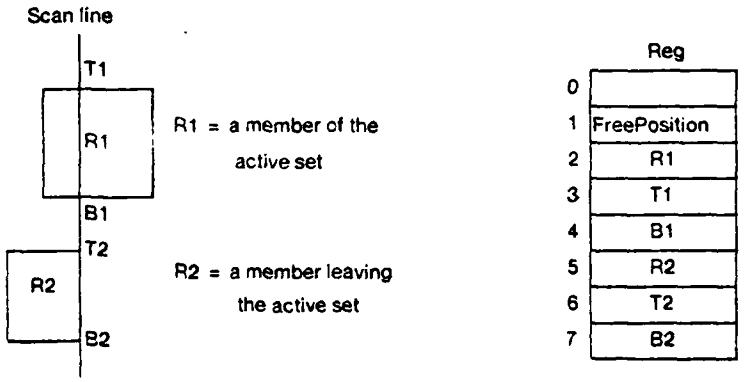
if it is directly above an odd number of edges or, in other words, a vertical scan line with one end tied to point P and the other to a weight intersects an odd number of edges. The singularity problem is shown in Figure 6-11 (c) where the scan line intersects the end points of some edges. At point A only one intersection should be counted, while at point B the count should be either none or two. Two solutions are mentioned in [76]. One solution is to displace slightly each polygon vertex that lies exactly on a scan line. A more reliable way is to use the topology of the polygon or the direction of successive edges of the polygon. If the polygon boundary progresses monotonically toward the left or right, only one intersection should be counted. On the other hand, if the direction changes from left to right or from right to left, then two intersections should be recorded.

The containment problem resembles the problem just examined in the last section. Here also, the problem involves a set of n line segments and a set of m points. A similar hardware-oriented solution follows. The solution, however, calls for a more complex design of the Δ-node which must also have some arithmetic capabilities. With a tree device capable of holding all the n edges, we can simply broadcast the m points in a pipelined fashion to the leaves where a test is made to verify if the edge lies below the point in question. If an output one is generated for an affirmative answer, then the final answer can be obtained by adding up the partial results in the internal nodes of the output tree. The problem can therefore be solved in O(m) time, after an initial O(n) loading phase.

If s < n but s ≥ m, then we can do the other way around by first loading the m points into the device and broadcast the n edges to the leaves. Each leaf will keep track of the number of intersections. This still takes O(m) + O(n) time. The case both n and m are greater than s can be handled by repeating the above procedure m/s times. The total time will be O(m n/s).

6.6 Performance Estimates

Consider the problem of detection of intersections of n rectangles (Section 6.3.2). Using the instruction set and timing estimates of Section 4.2, we compare the performance, with and without special-purpose hardware. Let T_{seq} and $T_{systolic}$ denote the time take by the sequential algorithm executing on a conventional computer and by the systolic tree device, respectively. We shall use k = 20 and s = 128. The two important phases are the initial sorting phase and the intersection detection phase. Given n rectangles, the hardware-oriented solution sorts $n^{1/2} / k$ collections of $2\,k\,n^{1/2}$ edges each. We now estimate the time needed per rectangle in the second phase. For every edge of the sorted vertical-edge sequence, the name of the corresponding rectangle, and the top and bottom y-values of the edge are sent to the interface controller, together with an indication of the type of the edge (whether left or right edge).


Scan line
T1
R1
B1
T2
R2
B2
R1 = a member of the active set
R2 = a member leaving the active set
Reg
0
1 FreePosition
2 R1
3 T1
4 B1
5 R2
6 T2
7 B2

(a) (b)

Figure 6-12: Nomenclature used and their respective assignment to register positions.

Depending on the type of the edge, the following sequence of instructions are issued. (Refer to Figure 6-12 (a) and (b) for the nomenclature used, as well as the corresponding register assignment.) If it is a left edge then the name of the rectangle R1, its top and bottom y-values T1 and B1, are inserted into the active set. We use the insertion mechanism as described earlier (Section 3.2.1.1). This is shown in Table 6-5.

Instruction	Comment
Input 0	Input position of free node
FreePosition	
Compare 1	See if selected
InputCond 2	Insert name of rectangle R1
R1	
InputCond 3	Insert top y-value T1
T1	
InputCond 4	Insert bottom y-value B1
B1	

Table 6-5: Actions fired by a left edge.

If the edge in question is a right edge, then the instructions of Table 6-6 are sent to the □-nodes and executed. These correspond to the actions of detection of intersections and the subsequent removal of the rectangle R2. Rectangle R2 intersects R1 if the following condition holds:

$$\neg(T1 < T2 \wedge T1 < B2 \vee B1 > T2 \wedge B1 > B2) \wedge R2 \neq R1.$$

The time spent for the processing of each rectangle is thus ($6\,N_a + 2\,N_b$) cycles, where N_a and N_b denote the number of input instructions and other instructions, respectively. With $N_a = 8$ and $N_b = 19$, this gives 86 cycles, or approximately 8.6 μs per rectangle. For each rectangle, four memory accesses are needed to fetch R1, T1, B1, and the edge type for the left edge, and similar quantities for the right edge. Thus strictly speaking, the three consecutive input instructions of Table 6-5 and Table 6-6 requires 3 memory cycles each. For a memory cycle of 800 ns, this would give approximately 9.2 μs per rectangle. The previously calculated value would still apply with some kind of buffering mechanism, to prefetch memory

Instruction	Comment
Input 5	Input name of rectangle R2
R2	
Input 6	Input top y-value T2
T2	
Input 7	Input bottom y-value B2
B2	
Load 3	$Flag_3 \leftarrow T1 < T2 \wedge T1 < B2$
Compare 6	
CompareAnd 7	
LoadFlag 1	
StoreFlag 3	
Load 4	$Flag_4 \leftarrow \neg(Flag_3 \vee B1 > T2 \wedge B1 > B2)$ $(Flag_4 \leftarrow R1 \cap R2)$
Compare 6	
CompareAnd 7	
LoadFlag 2	
Or 3	
Not 0	
StoreFlag 4	
Load 5	Restore node to free pool if R1 = R2
Compare 2	
InputCond 1	
NewFreePosition	
Not 0	$Flag_0 \leftarrow R1 \neq R2 \wedge R1 \cap R2$
And 4	
Output 2	Prepare to output R1 and R2
Output 5	
StoreFlag 7	

Table 6-6: Actions fired by a right edge.

data while a right edge is being handled. Alternately, we can sometimes shuffle input instructions with computations to reduce occurrence of consecutive input instructions.

Consider now the recently developed algorithm of [8]. We use an internal sorting method (Quicksort) to sort the 2 n edges, assuming we have enough memory. For the second phase, the reported time per rectangle is about 1.7 ms. We can now produce Table 6-7.

	s	n	T_{seq}	$T_{systolic}$	Speed-Up
1st phase	128	2048	983 ms	13.6 ms	72
		8192	4.59 s	54.2 ms	85
		131072	1 min 34 s	865 ms	109
		1048576	14 min 41 s	10.3 s	85
		4194304	1 h 4 min	41.1 s	94
		16777216	4 h 40 min	2 min 45 s	102
2nd phase	128		1.7 ms per rectangle	.009 ms per rectangle	177

Table 6-7: Performance estimates for the rectangle intersection problem.

6.7 Concluding Remarks

The special-purpose device can be of considerable help to speed up a sequential algorithm which depends heavily on sorting and other basic operations it supports. Obtaining such an algorithm may not be a trivial task, as shown by the three-dimensional matching problems. Also, it is crucial to start with an efficient sequential algorithm and try to speed it up by a special-purpose device.

In Chapter 5, we have concluded that the size s of a hardware sorting device can be small. The O(log s) upper bound on the speed-up does not encourage use of a large device. Here we have found another example where this is true. The locality property in the distribution of objects in certain design databases, together with an appropriate partitioning procedure, allows a small device to aid in detecting pairwise intersections of these objects.

Chapter 7
Conclusion

In this research we have investigated the feasibility of employing special-purpose VLSI hardware to solve compute-bound database problems. The device we have proposed can be viewed as a hardware implementation of a number of useful data structures where several basic operations are supported. Such a device has been shown to be useful in a large class of computationally intensive database problems. In addition to sorting, it can handle all the basic relational operations as join, project, union, and intersection. Furthermore, we have shown that a variety of database problems rely heavily on this same set of basic operations. These include some costly three-dimensional pattern and shape detection problems. Though presented with the purpose of illustrating the use of special-purpose hardware, solutions to some of the problems are interesting in their own right.

Research efforts on the design of systolic algorithms have concentrated on the achievement of high-performance through extensive use of parallelism and pipelining, inherent in the structure of the proposed algorithms. A special-purpose device is, however, one component among many other subsystems. It is time now to link the various subsystems together and start investigating important system issues. Limited size of special-purpose devices demands the partitioning of a large problem, which aggravates the I/O problem. Unfortunately, an s-fold parallelism does not necessarily imply an O(s) speed-up in relation to a sequential algorithm. We have shown that any special-purpose sorting hardware of size s can give at most O(log s) speed-up. The results on the expected performance improvements of the proposed systolic solutions have been found to be encouraging. For practical ranges of the quantity of data to be sorted, systolic devices of small sizes can beat even a recently developed fast linear algorithm.

The systolic architecture concept exploits the high component density promised by VLSI technological advances. Davis, Denny, and Sutherland [24] note that "topology has a major influence on whether or not a system can be implemented on a set of densely integrated VLSI chips". Our results show that in fact this is possible with tree structures such as the linearized tree and the binary tree. Not only can a subtree be compactly laid out on a component chip, but the component chips can themselves be laid out on PC boards in an identical manner. This result shows that a large tree can be packaged among chips in an economical way, with a constant pin requirement independent of the size of the tree. In terms of chip area per node, the custom design approach offers considerable savings with respect to the one-microprocessor-per-node approach where more powerful general-purpose microprocessors are used. In the custom design approach, the number of available pins limits the datapath width between nodes. A narrow datapath implies some scheme of serial data transmission. The performance of an individual node may compare unfavorably to that of a microprocessor. High performance in the custom VLSI approach is achieved through intensive use of parallelism. (Since the design costs are due mainly to the design of the basic cells, they are independent of the size of the systolic device.) With higher densities, greater degree of parallelism can be used, which in turn signifies higher performance. Therefore the systolic approach is likely to be cost effective.

7.1 Main Contributions and Results

The main contributions and results are summarized below.

- Many survey articles on database machine designs exist in the literature. They usually treat each individual design independently, with no effort dedicated to classify related works. A lot of seemingly different designs are often similar in conception, the main difference being in implementation details. The proposed taxonomy attempts to group the existing designs in a few categories. It facilitates the appraisal of existing designs whose strong and weak points are identified.

- We have presented a novel space allocation scheme which takes full advantage of pipelining and requires less logic than the previously proposed methods. Among the basic database operations, we have designed systolic algorithms for project, sort, join, union, etc. We have also discussed partitioning procedures for handling large problems.

- We have emphasized the importance of viewing special-purpose hardware at an overall system level and stressed the crucial role played by I/O and problem decomposition in the ultimate performance. The ideal special-purpose device design is one that achieves a balance between computation and I/O. A model to study the I/O complexity for sorting with special-purpose hardware devices has been proposed. A lower bound result on the I/O complexity for sorting n numbers with any special-purpose hardware device of size s is shown to be $\Omega(n \log n/\log s)$. An optimal design that achieves this bound has been presented. Based on fairly low-level considerations, we have estimated the performance improvements. We have also investigated the interface controller design for the proposed sorting device, which allows the overlap of computation and I/O.

- To evaluate the theme that a data structure supporting a few basic operations can be useful in many different database problems, we have examined the following problems all of which depend heavily on sort, join, and search operations.

  - Detection of three-dimensional patterns of m points in a large structure of n points (with application in chemical databases). We have presented two sequential algorithms whose worst-case time complexities are, under some mild assumptions, equal to $O(n^2 \log m)$ and $O(n^2)$. With a uniform distribution of points in the pattern, the time complexity is at most $O(m\ n)$. Compared with a previous result of order $\binom{n}{m}$, this new result is interesting in its own right.

  - Three-dimensional shape matching (with application in CAD databases). In addition to rotation, translation, and mirror image, scale is also allowed as a valid Euclidean transformation. This problem is thus more general than the previous one. The same algorithms for the previous problem are slightly modified to yield solutions with a worst-case time complexity of $O(n^2)$. This again constitutes a self-contained result.

  - Reporting intersection of rectangles with application in design databases. By exploiting the locality property in the distribution of rectangles in certain kinds of databases, we show that a relatively small special-purpose device can be used to aid in solving this problem.

  - Multidimensional searching with application in geographic and cartographic databases. In order to obtain fast query time in this searching problem, known sequential algorithms rely on preprocessing which is expensive both in time and storage. A straightforward hardware-oriented solution is shown.

  - The containment problem with application in design databases. Here again, the basic operation is that of search.

- We have proposed a new structure, called the *linearized tree*, that combines the characteristics of the binary tree and the linear array. A recursive procedure to obtain its compact layout has been shown. The layout result can be applied to the packaging of certain large tree structures among chips. These include the binary tree and the linearized tree. The packaging method requires only one type of fully utilized chip, which is not pin-bound. The wire lengths connecting component chips are shorter than those using the previously known layouts.

7.2 Further Work

The von Neumann bottleneck, associated with the communication path between the memory and the central processor, is a well known problem. It may ultimately dictate the overall performance of special-purpose chips. We feel that there is an urgent need for a formal model and techniques to investigate the bandwidth requirements in such special-purpose designs. Works such as [37] and this one, though each treating specific examples, may constitute the first steps toward such models.

A superficial study of the interface system has been carried out. The success of a special-purpose device depends heavily on the interface system. Questions as to what constitute the requirements of the interface system, or where the main activities are to take place, are vital to the success of the special-purpose approach and deserve further investigation.

We have concentrated on the packaging of certain tree structures into component chips. It may be interesting to investigate similar packaging problems with other structures. The linearized tree, whose layout result is used to solve the packaging problem of certain tree structures, is interesting in its own right. For example, Kung [54] has shown the correspondence between classes of efficient systolic algorithms with a number of communication topologies. It may be worthwhile to investigate the types of problems best suited for the proposed linearized tree structure. Along a more theoretic direction, it may be interesting to characterize the intrinsic improvement of the linearized tree structure over the binary tree. One starting point can the work by Hong and Rosenberg [38], who prove that the two structures are in fact equivalent under a certain cost measure. They consider embedding one target structure in another host structure, with the cost related to the largest distance in the host structure between images of adjacent vertices of the target structure. The measure is thus topological in nature. Other cost measures of geometric nature may be considered.

Having shown that a small systolic sorting device can offer substantial performance improvements over fast sequential sorting algorithms, the next step may be to build such a device. The linear array version probably should be used. However, there is still some distance between the present results on sorting and a concrete implementation. Among the studies yet to be made, we can cite the following. We have handled sorting of long keys by lexicographic sort. We need yet to investigate the best way to handle keys of variable length. Also, depending on the format of the raw data to be sorted, some transformation may be needed before they can be handled by the sorting device. If data transformation is to be carried out by the interface controller as each datum is fed into the sorting device, we should be very careful in the design so that the balance between I/O and computation is not disturbed. The multiway merge sort algorithm is very general and can be used for any distribution of the input data. If we have more knowledge on the data to be sorted (for example, they are "nearly" sorted, or uniformly distributed in a certain range), it may be worthwhile to develop other algorithms that would give a better average performance.

Appendix A
The Relational Data Model

We present a succinct introduction to the relational data model. A brief discussion of the implementation of relational database systems on a von Neumann machine is included at the end of this appendix. Details of the relational model can be found in [19, 21]. A survey of *relational database systems can be found in* [44].

A.1 Relations

Given sets D_1, D_2, ..., D_n (not necessarily distinct), R is a *relation* on these n sets if it is a set of ordered n-tuples $<d_1, d_2, ..., d_n>$ such that d_1 belongs to D_1, d_2 belongs to D_2, ..., d_n belongs to D_n. More concisely, R is a subset of the Cartesian product S_1 x S_2 x ... x S_n. Sets D_1, D_2, ..., D_n are the *domains* of R. The value n is the *degree* of R. The number of n-tuples (or simply tuples) in a relation is the *cardinality* of the relation.

FLIGHT

FLIGHT #	ORIGIN	DESTINATION
212	LA	Pitt
214	LA	Pitt
218	LA	Las Vegas
42	Pitt	LA
772	Las Vegas	LA
777	Las Vegas	Lake Tahoe
778	Las Vegas	Lake Tahoe

PILOT

PILOT NAME	CITY (currently in)
Smith	LA
Jones	Lake Tahoe
Johnson	Lake Tahoe
Davis	LA
Adams	Pitt
McDonald	Pitt

Figure 7-1: The relations FLIGHT and PILOT.

Figure 7-1 illustrates two relations called FLIGHT and PILOT. As the example illustrates, it is convenient to represent a relation as a table. Each row of the table represents a tuple of the relation. Rows are also referred to as *records*. Likewise, columns are also referred to as *attributes*. Referring back to the original definition, the domains of a relation have an ordering defined among them. If we were to rearrange the columns of the FLIGHT relation in some different order, the resulting table would be a different relation, mathematically speaking. However, if columns are referred to by their names and if the names are all distinct, this restriction can be relaxed. Thus column order, just as row order, can be treated as immaterial. It is important to notice the difference between domain, on the one hand, and attributes which are drawn from the domain, on the other. An attribute represents the *use* of a domain within a relation. In the relation FLIGHT, ORIGIN and DESTINATION are attributes drawn from the same domain, namely, a set of names of cities.

A.2 Keys

A *primary key* for a given relation is a single attribute or a combination of attributes whose values uniquely identify the tuples of the relation. FLIGHT # in the relation FLIGHT is a primary key. A relation may not possess a single-attribute primary key. However a relation is a set and does not contain duplicate tuples. Every relation will have some combination of attributes that have the unique identification property. The existence is guaranteed by the fact that at least the combination of all attributes is such a combination. Therefore every relation has a (possibly composite) primary key. It is usually assumed that the primary key is non-redundant, in the sense that none of its constituent attributes is superfluous. In case a relation possesses more than one primary key, we may arbitrarily choose one of the candidate keys as *the* primary key for the relation. An attribute of relation R_1 is a *foreign key* if it is not the primary key of R_1 but its values are values of the primary key of some relation R_2 (R_1 and R_2 not necessarily distinct).

A.3 Basic Relational Operations

For the union, intersection, and difference operations, the two relations concerned must be *union-compatible*, i. e., they must be of the same degree and the respective attributes drawn from the same domain. The relations being operated on will be referred to as argument relations.

Union

The union of two relations A and B is the set of all tuples belonging to either A or B, or both.

Intersection

The intersection of two relations A and B is the set of all tuples belonging to both A and B.

Difference

The difference between two relations A and B, in that order, is the set of all tuples belonging to A and not to B.

Select

Select constructs a subset of tuples within a relation for which a specified predicate is satisfied. The predicate is expressed as a Boolean combination of terms, each term being a comparison that can be established as true or false for a given tuple by inspecting that tuple in isolation.

Example

Select the tuples of relation FLIGHT where DESTINATION = "LA" or FLIGHT # > 600. The result is illustrated in Figure 7-2.

FLIGHT #	ORIGIN	DESTINATION
42	Pitt	LA
772	Las Vegas	LA
777	Las Vegas	Lake Tahoe
778	Las Vegas	Lake Tahoe

Figure 7-2: A selection example.

Project

Project applied to a relation over a specified combination of attributes gives a relation derived from the argument relation with the non-specified attributes eliminated (and also duplicate tuples eliminated).

Example

Project the FLIGHT relation over ORIGIN and DESTINATION. The result is illustrated in Figure 7-3.

ORIGIN	DESTINATION
LA	Pitt
LA	Las Vegas
Pitt	LA
Las Vegas	LA
Las Vegas	Lake Tahoe

Figure 7-3: A projection example.

Join

If two relations have a domain in common, they can be joined over that domain. The result of the join is a relation in which each tuple is formed by concatenating two tuples, one from each of the argument relations, such that the two tuples concerned have the same value in the common domain.

Example

Join the relations FLIGHT and PILOT over the attribute ORIGIN of FLIGHT and the attribute CITY of PILOT. The result is illustrated in Figure 7-4.

A.4 Implementation of Relational Database Systems

We now discuss the storage structures that have been used in implementing relational database systems. The discussion is based on a paper describing the history and evaluation of system R [2], an important relational database system. The storage structures to be discussed are. however, general in nature and not limited to this particular implementation.

One relational access method is called the XRM (Extended Relational Memory) developed by R. A. Lorie [71]. Each tuple of an n-ary relation has a TID (tuple identifier) which can be

FLIGHT #	ORIGIN	DESTINATION	PILOT NAME	CITY (currently in)
212	LA	Pitt	Smith	LA
214	LA	Pitt	Smith	LA
218	LA	Las Vegas	Smith	LA
212	LA	Pitt	Davis	LA
214	LA	Pitt	Davis	LA
218	LA	Las Vegas	Davis	LA
42	Pitt	LA	Adams	Pitt
42	Pitt	LA	McDonald	Pitt

Figure 7-4: A join example.

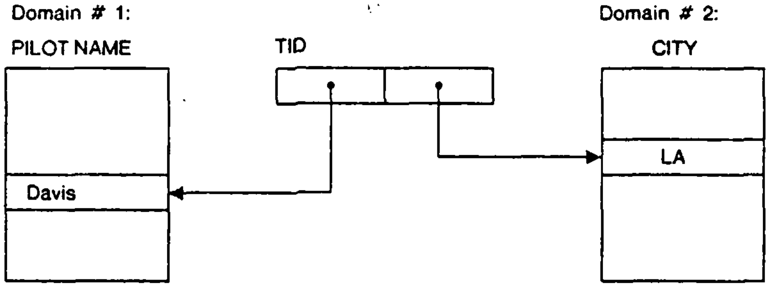


Figure 7-5: XRM storage structure.

used to fetch the associated tuple in one page reference. Rather than the actual data values, the tuple contains pointers to locations where the actual data items are stored (Figure 7-5 illustrates the representation of a tuple of the relation PILOT). Optionally, each domain may have an *inversion* which associates a given domain value with the TID's of tuples in which that value appears. In the example of Figure 7-5, if inversions exist on the domain CITY, then provisions can be made to create a list of TID's of pilots current in LA. A query can be processed by first manipulating the TID lists to obtain a list of TID's of tuples which satisfy the query, before any tuples are actually fetched. An advantage of the XRM is that commonly used values are represented only once. One serious drawback of XRM is that storing the domains separately from the tuples causes many extra I/O's to be done in retrieving data values. Because of this, the XRM was subsequently abandoned in a later implementation.

The access method in the newer implementation is called the RSS (Research Storage

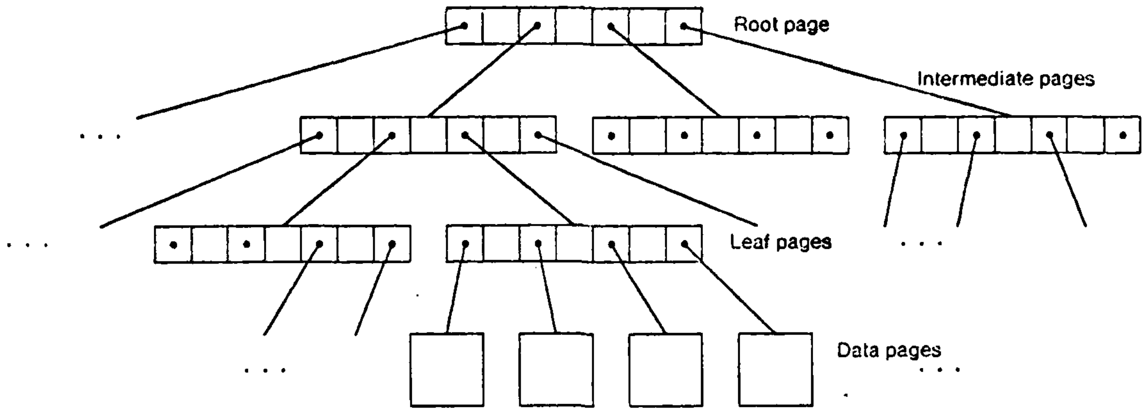


Figure 7-6: A B-tree index.

System). Rather than storing data values in separate domains, the RSS chose to store data values in the records of the database. Commonly used values are represented many times. It was felt that this disadvantage was more than overcome by the advantage that all the data values of a record can be fetched by a single I/O. Instead of the XRM inversions, the RSS uses *indexes* which are associative access aids implemented in the form of B-trees [22]. Each table in the database may have from zero indexes up to an index on each column. The B-tree index is the principal access path used in system R. An example is illustrated in Figure 7-6. Assuming a fan-out of 200 at each level of the tree, we can index a table of up to 8,000,000 records by a three-level index. Assuming that the root page remains in the main storage, three I/O's will typically be required to begin an associative scan through a large table. In addition to indexes, the RSS also implements *links* which are pointers stored with a record to connect it to related records in the same table or in another table. Another common access technique is hashing. Hashing was not used in System R because it does not have the convenient ordering property of a B-tree. Using indexes in preference to hashing, however, causes a performance penalty for transactions which access only one or two records. For example, to access a record in a large table using a three-level index, three I/O's are needed. If hashing is used, only one I/O may be sufficient, assuming no hash collisions occur.

References

[1] M. M. Astrahan et al.
System R: Relational Approach to Database Management.
ACM Transactions on Database Systems 1(2):97-137, June, 1976.

[2] M. M. Astrahan et al.
A History and Evaluation of System R.
Technical Report RJ 2843, IBM Research Laboratory, San Jose, California, December, 1980.

[3] E. Babb.
Implementing a Relational Database by means of Specialized Hardware.
ACM Transactions on Database Systems 4(1):1-29, March, 1979.

[4] H. S. Baird.
Fast Algorithms for LSI Artwork Analysis.
Design Automation and Fault-Tolerant Computing 2(2):179-209, May, 1978.

[5] J. Banerjee, D. K. Hsiao, and K. Kannan.
DBC - A Database Computer for Very Large Databases.
IEEE Transactions on Computers 28(6):414-429, June, 1979.

[6] J. Banerjee, D. K. Hsiao, and J. Menon.
The Clustering and Security Mechanisms of a Database Computer.
Technical Report OSU-CISRC-TR-79-2, The Ohio State University, Computer and Information Science Research Center, April, 1979.

[7] J. L. Bentley and W. Carruthers.
Algorithms for Testing the Inclusion of Points in Polygons.
18th Allerton Conference.

[8] J. L. Bentley, D. Haken and R. W. Hon.
Statistics on VLSI designs.
Technical Report CMU-CS-80-111, Carnegie-Mellon University, Computer Science Department, 1980.

[9] J. L. Bentley and H. T. Kung.
A Tree Machine for Searching Problems.
In *Proceedings of 1979 International Conference on Parallel Processing*, pages 257-266. IEEE, August, 1979.
Also available as a CMU Computer Science Department technical report CMU-CS-79-142, September, 1979.

[10] J. L. Bentley and T. Ottmann.
The Complexity of Manipulating Hierarchically Defined Sets of Rectangles.
Technical Report CMU-CS-81-109, Carnegie-Mellon University, Computer Science Department, April, 1981.

[11] K. J. Berkling.
A Computing Machine Based on Tree Structures.
IEEE Transactions on Computers C-20(4):404-418, April, 1971.

[12] R. R. Berman and M. Stonebraker.
A System for the Manipulation and Display of Geographic Data.
Computer Graphics 11(2):186-191, Summer, 1977.

[13] M. W. Blasgen and K. P. Eswaran.
On the Evaluation of Queries in a Relational Database System.
Technical Report RJ 1745, IBM Research Laboratory, San Jose, California, April, 1976.

[14] S. A. Browning.
Computations on a Tree of Processors.
In *Proc. Conference on Very Large Scale Integration: Architecture, Design, Fabrication*, pages 453-478. January, 1979.
Conference held at Caltech in Pasadena, California.

[15] S. A. Browning.
The Tree Machine: A Highly Concurrent Computing Environment.
PhD thesis, Computer Science Department, California Institute of Technology, January, 1980.

[16] A. K. Chandra and P. M. Merlin.
Optimal Implementation of Conjunctive Queries in Relational Data Bases.
In *Proceedings of Ninth Annual ACM Symposium on Theory of Computing*, pages 77-90. May, 1977.

[17] H. Chang.
On Bubble Memories and Relational Data Base.
In *Proceedings 4th International Conference on Very Large Data Bases*, pages 207-229. 1978.

[18] T. C. Chen, V. W. Lum, and C. Tung.
The Rebound Sorter: An Efficient Sort Engine for Large Files.
In *Proceedings 4th International Conference on Very Large Data Bases*, pages 312-318. September, 1978.

[19] E. F. Codd.
A Relational Model of Data for Large Shared Data Banks.
Communications of the ACM 13(6):377-387, June, 1970.

[20] E. F. Codd.
Further Normalization of the Data Base Relational Model.
In *Courant Computer Science Symposia, Vol. 6: Data Base Systems*. Prentice-Hall, 1971.

[21] E. F. Codd.
Relational Completeness of Data Base Sublanguages.
In *Courant Computer Science Symposia, Vol. 6: Data Base Systems*. Prentice-Hall, 1972.

[22] D. Comer.
The Ubiquitous B-Tree.
ACM Computing Surveys 11(2):121-137, June, 1979.

[23] G. F. Coulouris, J. M. Evans, and R. W. Mitchell.
Towards Content Addressing in Data Bases.
Computer Journal 15(2):95-98, May, 1972.

[24] A. L. Davis, W. M. Denny, and I. Sutherland.
A Characterization of Parallel Systems.
Technical Report UUCS-80-108, Department of Computer Science, University of Utah, August, 1980.

[25] D. J. DeWitt.
DIRECT - A Multiprocessor Organization for Supporting Relational Database Management Systems.
IEEE Transactions on Computers C-28(6):395-406, June, 1979.

[26] D. Dobkin and R. J. Lipton.
Multidimensional Searching Problems.
SIAM Journal of Computing 5(2):181-186, June, 1976.

[27] C. M. Eastman and R. Thornton.
A Report on the GLIDES2 Language Definition.
Technical Report, Institute of Physical Planning, Department of Architecture, Carnegie-Mellon University, 1979.

[28] R. Fagin.
Multivalued Dependencies and a New Normal Form for Relational Databases.
ACM Transactions on Database Systems 2(3):262-278, September, 1977.

[29] R. Fagin.
On an Authorization Mechanism.
ACM Transactions on Database Systems 3(3):310-319, September, 1978.

[30] A. L. Fisher.
Systolic Algorithms for Running Order Statistics in Signal and Image Processing.
Technical Report, Carnegie-Mellon University, Computer Science Department, 1981.
In preparation.

[31] M. J. Foster and H. T. Kung.
The Design of Special-Purpose VLSI Chips.
Computer 13(1):26-40, January, 1980.
A condensed version appears in *Conference Proceedings of the 7th Annual Symposium on Computer Architecture*, May 1980, pp. 300-307.

[32] M. J. Foster.
Tools for Generating VLSI Language Transducers: a Thesis Proposal.
VLSI Document V083, Carnegie-Mellon University, Computer Science Department, April, 1981.

[33] D. Haken.
A Geometric Design Rule Checker.
VLSI Document V053, Carnegie-Mellon University, Computer Science Department, June, 1980.

[34] C. A. R. Hoare.
Quicksort.
Comp. J. 5:10-15, 1962.

[35] L. A. Hollaar.
Specialized Merge Processor Networks for Combining Sorted Lists.
ACM Transactions on Database Systems 3(3):272-284, September, 1978.

[36] R. Hon.
The Hierarchical Analysis of VLSI Designs: a Thesis Proposal.
VLSI Document V073, Carnegie-Mellon University, Computer Science Department, December, 1980.

[37] J. W. Hong and H. T. Kung.
I/O Complexity: The Red-Blue Pebble Game.
In *Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing*, pages 326-333. May, 1981.
Also available as a CMU Computer Science Department technical report CMU-CS-81-111, March, 1981.

[38] J. W. Hong and A. L. Rosenberg.
Graphs that are Similar to Binary Trees.
Technical Report RC 8402, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, July, 1980.

[39] D. K. Hsiao and M. J. Menon.
The Post Processing Functions of a Database Computer.
Technical Report OSU-CISRC-TR-79-6, Computer and Information Science Research Center, The Ohio State University, July, 1979.

[40] D. K. Hsiao and M. J. Menon.
Parallel Record Sorting Methods for Hardware Realization.
Technical Report OSU-CISRC-TR-80-7, Computer and Information Science Research Center, The Ohio State University, July, 1980.

[41] D. K. Hsiao and M. J. Menon.
Design and Analysis of Relational Join Operations of a Database Computer (DBC).
Technical Report OSU-CISRC-TR-80-8, Computer and Information Science Research Center, The Ohio State University, September, 1980.

[42] Y. E. Kalay and C. M. Eastman.
Shape Operations - an Algorithm for Spatial-Set Manipulations of Solid Objects.
Technical Report, CAD - Graphic Laboratory, Institute of Building Sciences, Department of Architecture, Carnegie-Mellon University, 1980.

[43] Won Kim, D. J. Kuck, and D. Gajski.
A Bit-Serial/Tuple-Parallel Relational Query Processor.
Technical Report RJ 3194, IBM Research Laboratory, San Jose, California, July, 1981.

[44] Won Kim.
Relational Database Systems.
ACM Computing Surveys 11(3):185-211, September, 1979.

[45] D. E. Knuth.
The Art of Computer Programming. Volume 1: *Fundamental Algorithms.*
Addison-Wesley, Reading, Massachusetts, 1973.

[46] D. E. Knuth.
The Art of Computer Programming. Volume 3: *Sorting and Searching.*
Addison-Wesley, Reading, Massachusetts, 1973.

[47] I. Kobayashi.
Cartographic Databases.
In S. K. Chang and K. S. Fu (editors), *Pictorial Information Systems*, pages 322-350. Springer-Verlag, 1980.

[48] P. M. Kogge.
Maximal Rate Pipelined Solutions to Recurrence Problems.
Unpublished report.

[49] H. T. Kung, L. J. Guibas, and C. D. Thompson.
Direct VLSI implementation of combinatorial algorithms.
In *Proc. Conference on Very Large Scale Integration: Architecture, Design, Fabrication*, pages 509-525. California Institute of Technology, January, 1979.

[50] H. T. Kung and P. L. Lehman.
Systolic (VLSI) Arrays for Relational Database Operations.
In *Proceedings of the ACM SIGMOD 1980 International Conference on Management of Data*, pages 105-116. ACM, May, 1980.
Conference held in Santa Monica, California. Also available as a Carnegie-Mellon University Computer Science Department technical report CMU-CS-80-114, March, 1980.

[51] H. T. Kung and C. E. Leiserson.
Systolic Arrays (for VLSI).
In Duff, I. S. and Stewart, G. W. (editors), *Sparse Matrix Proceedings 1978*, pages 256-282. SIAM, 1979.
A slightly different version appears in *Introduction to VLSI Systems* by C. A. Mead and L. A. Conway, Addison-Wesley, 1980, Section 8.3.

[52] H. T. Kung and R. L. Picard.
Hardware Pipelines for Multi-Dimensional Convolution and Resampling.
Workshop on Computer Architecture for Pattern Analysis and Image Database Management, November, 1981.

[53] H. T. Kung and S. W. Song.
A Systolic 2-D Convolution Chip.
Technical Report CMU-CS-81-110, Carnegie-Mellon University, Computer Science Department, March, 1981.
To appear in *Non-Conventional Computers and Image Processing: Algorithms and Programs*, Leonard Uhr (editor), Academic Press, 1981.

[54] H. T. Kung.
Let's Design Algorithms for VLSI Systems.
In *Proc. Conference on Very Large Scale Integration: Architecture, Design, Fabrication*, pages 65-90. January, 1979.
Conference held at Caltech in Pasadena, California. Invited paper.

[55] H. T. Kung.
Special-Purpose Devices for Signal and Image Processing: An Opportunity in VLSI.
In *Proceedings of the SPIE, Vol. 241, Real-Time Signal Processing III*, pages 76-84. The Society of Photo-Optical Instrumentation Engineers, July, 1980.

[56] H. T. Kung.
Why Systolic Architecture.
To appear in *Computer Magazine*, 1981.

[57] G. M. E. Lafue.
Design Data Base and Data Base Design.
Technical Report Research Report No. 74, Institute of Physical Planning, Department of Architecture, Carnegie-Mellon University, March, 1978.
This paper was presented at the CAD78 Conference held in Brighton, England, March 1978.

[58] G. M. E. Lafue.
An Approach to Automatic Maintenance of Semantic Integrity in Large Design Databases.
Technical Report, School of Urban and Public Affairs, Carnegie-Mellon University, 1979.
Paper presented at the IFIPS National Computer Conference, 1979.

[59] T. Lang, E. Nahouraii, K. Kasuga, and E. B. Fernandez.
An Architectural Extension for a Large Database System Incorporating a Processor for Disk Search.
In *Proceedings of the Third International Conference on Very Large Data Bases*, pages 204-210. 1977.

[60] G. G. Langdon Jr.
A Note on Associative Processors for Database Management.
ACM Transactions on Database Systems 3(2):148-158, June, 1978.

[61] D. T. Lee and F. P. Preparata.
Location of a Point in a Planar Subdivision and its Applications.
SIAM Journal of Computing 6(3):594-606, September, 1977.

[62] P. L. Lehman.
The Theory and Design of Systolic Database Machines.
Thesis Proposal. Carnegie-Mellon University, Computer Science Department, December, 1980.

[63] H. O. Leilich, G. Stiege, and H. C. Zeidler.
A Search Processor for Data Base Management Systems.
In *Proceeding 4th Conference on Very Large Data Bases*, pages 280-287. September, 1978.

[64] C. E. Leiserson.
Systolic Priority Queues.
In *Proc. Conference on Very Large Scale Integration: Architecture, Design, Fabrication*, pages 199-214. Caltech, January, 1979.
Also available as a CMU Computer Science Department technical report CMU-CS-79-115, April, 1979.

[65] C. E. Leiserson.
Area-Efficient VLSI Computation.
PhD thesis, Carnegie-Mellon University, Computer Science Department, 1981.

[66] M. P. Lepselter.
X-Ray Lithography Breaks the Submicrometer Barrier.
IEEE Spectrum 18(5):26-29, May, 1981.

[67] A. M. Lesk.
Detection of Three-Dimensional Patterns of Atoms in Chemical Structures.
Communications of the ACM 22(4):219-224, April, 1979.

[68] C. S. Lin, D. C. P. Smith, and J. M. Smith.
The Design of a Rotating Associative Memory for Relational Database Applications.
ACM Transactions on Database Systems 1(1):53--65, March, 1976.

[69] C. S. Lin.
Sorting with Associative Secondary Storage Devices.
In *Proceedings of the National Computer Conference*, pages 691-695. 1977.

[70] Clive Liu.
Object Grammar - Language on the Generation of Masonry Design.
Technical Report, Institute of Building Sciences, Department of Architecture, Carnegie-Mellon University, June, 1981.

[71] R. A. Lorie.
XRM - An Extended (N-ary) Relational Memory.
Technical Report G320-2096, IBM Scientific Center, Cambridge, Ma., January, 1974.

[72] G. A. Magó.
A Network of Microprocessors to Execute Reduction Languages, Part I.
International Journal of Computer and Information Sciences 8(5):349-385, March, 1979.

[73] G. A. Magó.
A cellular Computer Architecture for Functional Programming.
In *COMPCON Spring 1980.* IEEE, 1980.

[74] C. A. Mead and L. A. Conway.
Introduction to VLSI Systems.
Addison-Wesley, Reading, Massachusetts, 1980.

[75] C. Mead and M. Rem.
Cost and Performance of VLSI Computing Structures.
IEEE Journal of Solid State Circuits SC-14(2):?, April, 1979.

[76] W. M. Newman and R. F. Sproull.
Principles of Interactive Computer Graphics.
McGraw-Hill, 1979.

[77] M. Rem.
Interactions of Hardware and Software for Associative Processors.
Technical Report, California Institute of Technology, Division of Engineering and Applied Science, June, 1978.

[78] A. L. Rosenberg, D. Wood, and Z. Galil.
Storage Representations for Tree-Like Data Structures.
Mathematical Systems Theory 13(2):105-130, 1979.

[79] S. A. Schuster, H. B. Nguyen, E. A. Ozkarahan, and K. C. Smith.
RAP 2 - An Associative Processor for Databases and its Applications.
IEEE Transactions on Computers C-28(6):446-458, June, 1979.

[80] C. H. Sequin, A. M. Despain, and D. A. Patterson.
Communication in X-tree, a Modular Multiprocessor System.
In *ACM Proceedings 1978 Annual Conference*, pages 194-203. December, 1978.

[81] M. I. Shamos and D. Hoey.
Geometric Intersection Problems.
In *Proceedings of the 17th Annual Symposium on Foundations of Computer Science*, pages 208-215. IEEE, October, 1976.

[82] M. I. Shamos.
Computational Geometry.
PhD thesis, Yale University, May, 1978.

[83] D. E. Shaw.
A Hierarchical Associative Architecture for the Parallel Evaluation of Relational Algebraic Database Primitives.
Technical Report STAN-CS-79-778, Department of Computer Science, Stanford University, October, 1979.

[84] D. L. Slotnick.
Logic per Track Devices.
In Tou, J. (editor), *Advances in Computers, Vol. 10*, pages 291-296. Academic Press, New York, 1970.

[85] S. W. Song.
A Highly Concurrent Tree Machine for Database Applications.
In *Proceedings of the 1980 International Conference on Parallel Processing*, pages 259-268. IEEE, August, 1980.
Also available as a CMU technical report, VLSI Document V055, June, 1980.

[86] G. Stiny.
Pictorial and Formal Aspects of Shape and Shape Grammars.
PhD thesis, System Science Department, University of California, Los Angeles, 1975.

[87] G. Stiny.
Two Exercises in Formal Composition.
Environment and Planning B 3:187-210, 1976.

[88] G. Stiny.
Ice-Ray: a note on the Generation of Chinese Lattice Designs.
Environment and Planning B 4:89-98, 1977.

[89] H. S. Stone.
Parallel Processing with the Perfect Shuffle.
IEEE Transactions on Computers C-20(2):153-161, February, 1971.

[90] M. Stonebraker.
Implementation of Integrity Constraints and Views by Query Modification.
In *Proceedings of the ACM SIGMOD 1975 International Conference on Management of Data*, pages 65-78. May, 1975.

[91] S. Y. W. Su, L. H. Nguyen, A. Emam, and G. L. Lipovski.
The Architectural Features and Implementation Techniques of the Multicell CASSM.
IEEE Transactions on Computers C-28(6):430-445, June, 1979.

[92] B. W. Weide.
Statistical Methods in Algorithm Design and Analysis.
PhD thesis, Carnegie-Mellon University, Computer Science Department, August, 1978.

[93] T. Whitney.
Description of the Hierarchical Design Rule Filter.
Technical Report SSP File 4027, Computer Science Department, California Institute of Technology, October, 1980.

[94] P. Wilcox, H. Rombeek, and D. M. Caughey.
Design Rule Verification Based on One Dimensional Scans.
In *Proceedings of the Fifteenth Design Automation Conference*, pages 285-289. June, 1978.

[95] W. Wilner.
Recursive Machines.
Technical Report, XEROX Palo Alto Research Center, January, 1978.

[96] M. M. Zloof.
Query by Example.
In *Proceedings of the AFIPS National Computer Conference*, pages 431-438. 1975.

Index

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER CMU-CS-81-142	2. GOVT ACCESSION NO. AD-A112 542 3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) ON A HIGH-PERFORMANCE VLSI SOLUTION TO DATABASE PROBLEMS	5. TYPE OF REPORT & PERIOD COVERED Interim 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Siang Wun Song	8. CONTRACT OR GRANT NUMBER(s) N00014-76-C-0370 N00014-80-C-0236
9. PERFORMING ORGANIZATION NAME AND ADDRESS Carnegie-Mellon University Computer Science Department Pittsburgh, PA. 15213	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS	12. REPORT DATE August 1981 13. NUMBER OF PAGES 170
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	15. SECURITY CLASS. (of this report) UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

[illegible]

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Approved for public release; distribution unlimited

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)